Title
Cooperative Power and Resource Management for Heterogeneous Mobile Architectures

Author
Hsieh, Chenying

Publication Date
2019

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA,
IRVINE

Cooperative Power and Resource Management for Heterogeneous Mobile Architectures

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Chenying Hsieh

Dissertation Committee:
Professor Nikil Dutt, Chair
Professor Tony Givargis
Professor Ardalan Amiri Sani

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to express great gratitude to my advisor, Professor Nikil Dutt who has been my role model in both academia and life. He is patient, supportive and encouraging to pursue my goals and his passion of research always inspires me. I would not have finished my PhD without his guidance.

I want to thank Professor Sung-soo Lim and Professor Ardalan Amiri Sani for their professional and insightful comments about our work. The time spent working with them is really enlightening and enjoyable.

I am grateful to my colleagues and friends at UCI for all their support. In particular, I thank Professor Amir Rahmani, Dr. Santanu Sarma, Dr. Hossein Tajik, Dr. Majid Namaki Shoushtari, Dr. Jurngyu Park, Dr. Tiago Muck, Dr. Bryan Donyanavard, Dr. Zhi Chen, Dr. Ting-Shuo Zhou, Kasra Moazzemi, Hamid Nejatollahi, Biswadip Maity, Sina Labbaf, Saehanseul Yi, Michael Tao-Yi Lee and Jun-Wei Lin for the inspiring discussions, friendship and collaborations.

I deeply thank my families for their support, love and sacrifice. My parents always encourage me to pursue higher education and support me unconditionally. My sister Andrea helped me a lot settle down when I first arrived California and is caring for me without doubt. My girl friend Hanna is very understanding and has prepared food for me during my last year of PhD to let me focus on completing the thesis.

I acknowledge the Springer and IEEE for the permission to include my previously published work in [2], [3] and [4].

CURRICULUM VITAE

Chenying Hsieh

EDUCATION

Doctor of Philosophy in Computer Science **2019**
University of California, Irvine *Irvine, CA*

Master of Science in Computer Science **2008**
National Tsing Hua University *Hsinchu, Taiwan*

Bachelor of Science in Computer Science and Engineering **2006**
National Tsing Hua University *Hsinchu, Taiwan*

WORK AND RESEARCH EXPERIENCE

Graduate Research Assistant **2013–2019**
University of California, Irvine *Irvine, CA*

Software Engineering Intern **2016–2017**
Levyx Inc. *Irvine, CA*

System Software Engineer **2008–2012**
Sunplus Technology Co., Ltd. Inc. *Hsinchu, Taiwan*

Student Reseracher **2006–2008**
National Tsing Hua University *Hsinchu, Taiwan*

TEACHING EXPERIENCE

Teaching Assistant **2014–2019**
University of California, Irvine *Irvine, CA*

REFEREED JOURNAL PUBLICATIONS

MEMCOP: memory-aware co-operative power management governor for mobile games **2018**
Design Automation for Embedded Systems

REFEREED CONFERENCE PUBLICATIONS

SURF: Self-aware Unified Runtime Framework for Parallel Programs on Heterogeneous Mobile Architectures **Oct 2019**
27th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)

The Case for Exploiting Underutilized Resources in Heterogeneous Mobile Architectures **Mar 2019**
2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)

HAMEX: heterogeneous architecture and memory exploration framework **Oct 2016**
27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype (RSP)

Co-Cap: Energy-efficient Cooperative CPU-GPU Frequency Capping for Mobile Games **Apr 2016**
31st Annual ACM Symposium on Applied Computing (SAC)

Memory-aware cooperative CPU-GPU DVFS governor for mobile games **Oct 2015**
13th IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia)

Quality-aware Mobile Graphics Workload Characterization for Energy-efficient DVFS Design **Oct 2014**
12th IEEE Symposium on Embedded Systems for Real Time Multimedia (ESTIMedia)

SOFTWARE

SURF `https://github.com/chenying-hsieh/hmp-rt`
Unified Runtime Framework for Heterogeneous Mobile Architectures

ABSTRACT OF THE DISSERTATION

Cooperative Power and Resource Management for Heterogeneous Mobile Architectures

By

Chenying Hsieh

Doctor of Philosophy in Computer Science

University of California, Irvine, 2019

Professor Nikil Dutt, Chair

Heterogeneous architectures have been ubiquitous in mobile system-on-chips (SoCs). The demand from different application domains such as games, computer vision and machine learning which requires massive parallelism of computation has driven the integration of more accelerators into mobile SoCs to provide satisfactory performance energy-efficiently. These on-chip computing resources typically have their individual runtime systems including: (1) a software governor: continuously monitors hardware utilization and makes decisions of trade-off between performance and power consumption. (2) software stack: allows application developers to program the hardware for general purpose computation and perform memory management and profiling. As computation of mobile applications may demand all sorts of combinations of computing resources, we identify two problems: (1) individual runtime can often lead to poor performance-power trade-off or inefficient utilization of computing resources. (2) existing approaches fail to schedule subprograms among different computing resources and further lose the opportunity to avoid resource contention to gain better performance.

To address these two issues, we propose a holistic approach to coordinate different runtime regrading application performance and energy efficiency in this dissertation. We first present MEMCOP, a memory-aware collaborative CPU-GPU DVFS governor that considers both

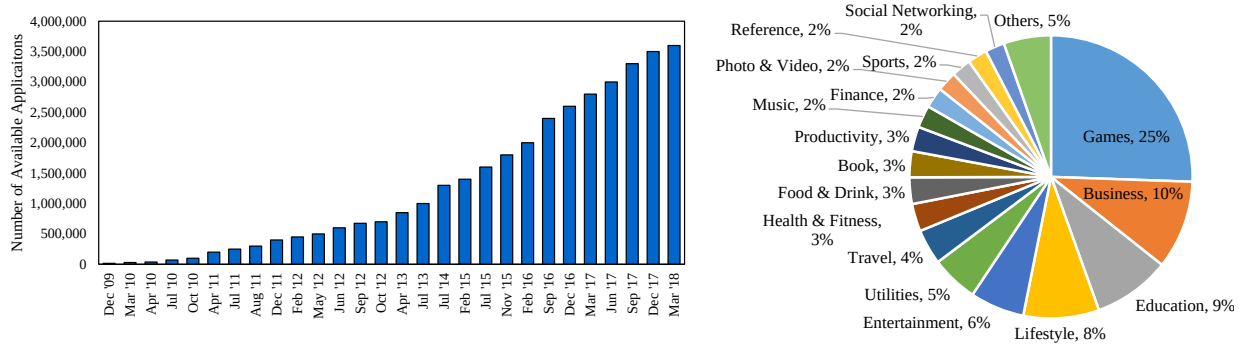
the memory access footprint as well as the CPU/GPU frequency to improve energy efficiency of high-end mobile game workloads by performing dynamic voltage and frequency scaling (DVFS). Second, we present a case study executing a mix of popular data-parallel workloads such as convolutional neural networks (CNNs), computer vision filters and graphics rendering kernels on mobile devices, and show that both performance and energy consumption of mobile platforms can be improved by synergistically deploying these underutilized compute resources. Third, we present SURF: a Self-aware Unified Runtime Framework for Parallel Programs on Heterogeneous Mobile Architectures. SURF supports several heterogeneous parallel programming languages (including OpenMP and OpenCL), and enables dynamic task-mapping to heterogeneous resources based on runtime measurement and prediction. The measurement and monitoring loop enables self-aware adaptation of run-time mapping to exploit the best available resource dynamically. We implemented all the software components on real-world mobile SoCs and evaluated our proposed approaches with mobile games and mix of parallel benchmarks and CNN applications.

Chapter 1

Introduction

1.1 Emerging Mobile Heterogeneous Architectures

Mobile devices were invented as portable devices with limited functions such as feature phones, designed to provide phone call and text messaging functionality in addition to limited capability of multi-media playback and internet access. Since the emergence of first Apple iPhone [5] published in 2007, and Google Android phone [6] in 2008, mobile devices have evolved to powerful computer systems and prevailed in the markets. The usage of mobile



(a) Number of available applications in Google Play Store [7]

(b) Application categories in Apple Store [8]

Figure 1.1: Mobile Applications Trends

devices such as mobile phone and tablets also have long surpassed desktops [9] and is still growing as shown by the growth in the available applications in Google Play Store [10] (Figure 1.1a) which is one of the most popular mobile application markets. Figure 1.1b shows the top applications categories in Apple Store, where the game category has dominated the market. There are also emerging mobile computer vision and machine learning applications which require massive parallelism. These applications have stimulated the development of more powerful and new hardware accelerators. For example, the high volume of game

(a) Qualcomm Snapdragon SoC graphics performance over time [11]

(b) Qualcomm Hexagon DSP performance for Tensorflow [12]

Figure 1.2: Trend of hardware accelerators in Mobile SoCs

applications and their need of high quality graphics necessitate GPU performance to improve continuously. Figure 1.2a shows that the Qualcomm Snapdragon graphics performance over time which has shown drastically performance improvement in the last decade. Similar trends are also found in other GPUs such as ARM Mali GPU [13] and Apple GPU. Likewise, new hardware accelerators such as Google Tensor Processing Unit (TPU) [14], Apple Neural Engine and Qualcomm Hexagon DSP [15] to energy-efficiently process the computation from these applications. Figure 1.2b shows Qualcomm Hexagon DSP is much more energy efficient than CPU and GPU for machine learning applications.

Battery-powered modern mobile devices are typically deployed with heterogeneous multi-processor system-on-chip (HMPSoC) which integrates one or more hardware accelerators such as GPU and DSP in order to provide satisfactory performance and energy efficiency. Figure 1.3 shows a typical architecture of a modern HMPSoC. The SoC integrates GPU and

DSP to accelerate graphics rendering and signal processing respectively. Besides, they also support general purpose computing which allows application developers to execute application blocks which are traditionally executed on CPU.

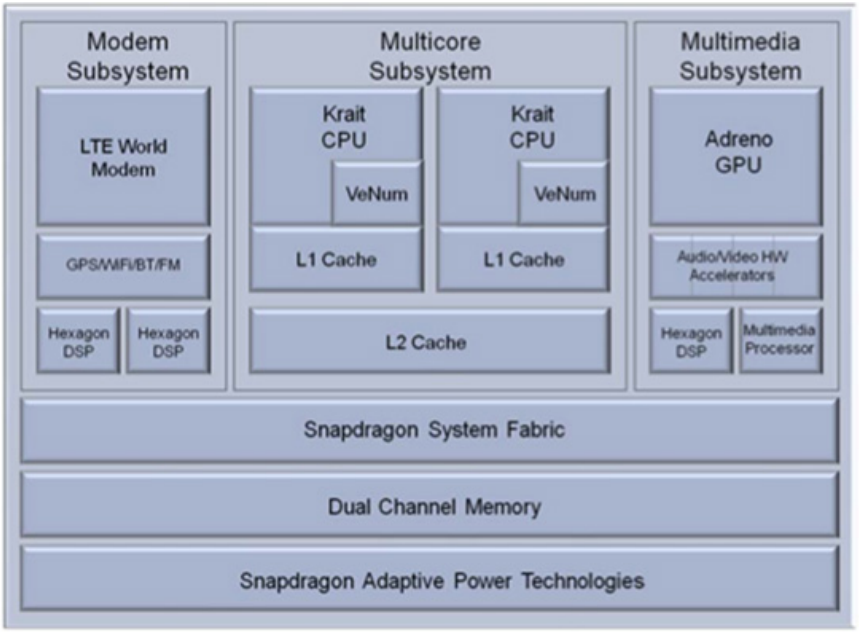


Figure 1.3: Example of modern mobile HMPSoC architecture [1]

HMPSoCs have a runtime system that performs power and resource management for each compute resources. The software components of a runtime system includes:

1. Governor: performs dynamic voltage and frequency scaling (DVFS) to adjust power consumption and device utilization in terms of active clock cycles versus total clock cycles. For example, Android Linux kernel provides formal CPU frequency governors such as ONDEMAND and INTERACTIVE governors which scale frequency according to CPU utilization. On the other hand, the governors of other hard accelerators are vendor-specific with no formalization.

2. Software stack: allows application developers to program the computing resources. The software stack is typically composed of programming interface, runtime library and device driver. Programming interface refers to frameworks such as OpenCL [16],

CUDA, and OpenMP. Runtime library and device driver are vendor-specific and provided by vendors which process requests from applications and deploy tasks to the hardware.

Figure 1.4 shows a high-level view of the software architecture of governor and the software stack.

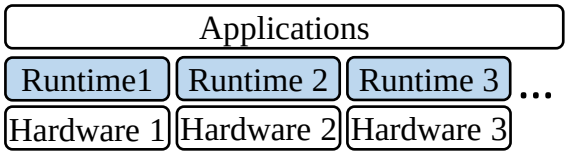


Figure 1.4: High-level view of parallel programming language architecture

1.2 Motivation and Challenges

Mobile SoCs inevitably need to address the trade-off between performance and power. Besides, the management in modern mobile SoCs faces a number of other challenges to exploit resources energy-efficiently and provide acceptable performance.

- Memory Contention: as shown in Figure 1.3, system memory is shared with all compute resources in modern mobile SoCs. It eliminates the overhead of moving data between CPU and accelerators which is significant as the computation executed in accelerators usually involves high bandwidth of memory traffic. The downside is while the accelerators are executing, the high memory traffic can lead of performance downgrade to other compute resources.

- Resource Contention and Underutilized Resources: applications developers are not limited to accelerate their applications in specific compute resource. Hence, when multiple applications attempt to execute on the same compute resource, the contention

can lead to performance downgrade. On the other hand, there can be idle compute resources while certain ones are already saturated. This opens up an opportunistic to execute on idle resources to provide better responsiveness of applications but may not be necessarily energy-efficiency.

- Individual Runtime System: modern mobile applications can easily accelerate application blocks by hardware accelerators. Hence, the performance of the applications depends on multiple compute resources, not only CPU traditionally. The runtime system should be aware of the computation composition of applications in order to provide energy efficiency.

1.3 Thesis Overview

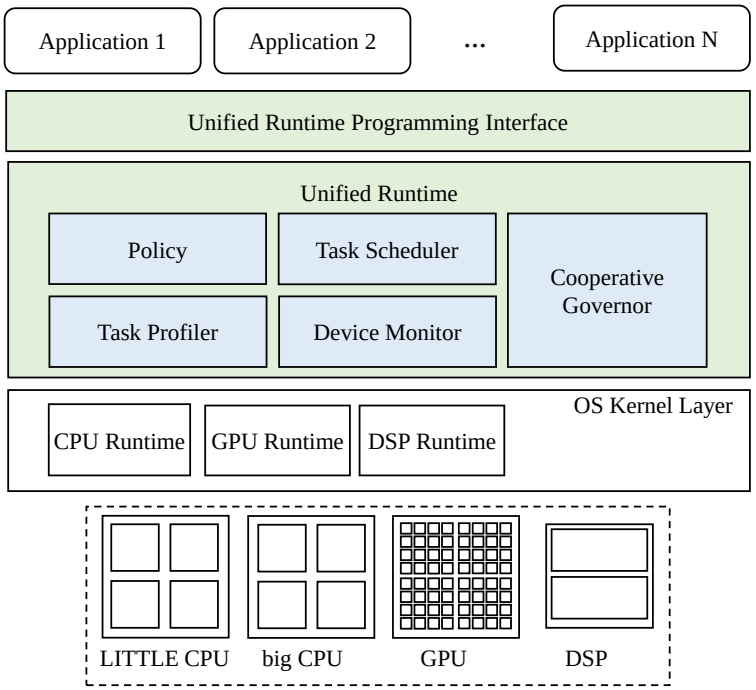


Figure 1.5: Overview of proposed unified runtime architecture

An intelligent management for mobile SoCs should address a number of challenges: (1) satisfactory performance (2) energy-efficiently utilizing computing resources. As emerging

resource-intensive mobile applications require acceleration from different hardware components, current individual runtime systems fail to respond to the challenges. The thesis demonstrates techniques to enable collaboration between different runtime system in HMP-SoCs. Figure 1.5 shows an overview of our contributions in this thesis toward collaborative power and resource management. In Chapter 2, we present MEMCOP, a memory-aware cooperative CPU-GPU governor for mobile game applications. Mobile games can generate high memory traffic when rendering high-quality graphics. As CPU and GPU share system memory, this memory traffic can affect CPU performance. This performance impact is not considered in the individual default CPU and GPU governors which results in poor energy efficiency for mobile games. MEMCOP continuously tracks the characteristics of CPU workload and system memory traffic frame by frame at runtime and make DVFS decisions according to performance models built offline. In Chapter 3, we present a case study of energy-efficiently executing a mix of data-parallel workload including convolutional neural network (CNN), computer vision filter and graphics rendering applications. These applications require computation with massive parallelism which is usually dealt by GPU. Hence, resource contention happens when multiple applications attempt to execute on GPU. This work shows the opportunity to achieve better performance and energy efficiency by collaboratively deploying underutilized compute resources. In chapter 4, we argue that current mobile systems are unable to schedule subprograms between compute units due to lack of vendor support and present SURF, a Self-aware Unified Runtime Framework for Parallel Programs on Heterogeneous Mobile Architectures. SURF provides interfaces for users to program compute resource with different programming interfaces such as OpenCL and OpenMP. SURF is able to monitor the utilization of all compute units and profile tasks. The core of SURF is an adaptive heterogeneous-earliest-finish-time (HEFT) task scheduler which allocate compute resource which has the earliest finish time according to the task profile and status of compute units. In Chapter 5, we discuss future directions and conclude the thesis.

Chapter 2

MEMCOP: Memory-Aware Co-operative Power Management Governor for Mobile Games

2.1 Introduction

Contemporary mobile SoCs are designed with heterogeneous architectures that integrate high-performance CPU, GPU and other IP blocks. Figure 2.1 shows a simplified block diagram of a recent mobile platform, Samsung Exynos 5422 Octa-core SoC. Such integration enables mobile graphics applications like games to provide high-quality graphics comparable to desktop graphics applications. However, these high quality graphics applications require high power and energy consumption leading to faster battery drain than ever. Dynamic power management (DPM) is commonly used to control the power consumption in mobile systems. The power manager for each component in mobile systems is called a governor. CPU, GPU, and memory systems usually have their dedicated governors considering the com-

ponents' characteristics. Some governors are architecturally optimized by the chip vendors considering both performance and power consumption features. Mobile operating systems still have not provided frameworks to synergize governors across applications to exploit multiple resources in mobile systems. Many research efforts have tried to solve the problems of the governors to significantly improve the battery life in mobile systems. However, most of the previous efforts focus on only a single governor to solve the high energy consumption while running mobile games without considering integrated effects of the governors. Recent efforts [17] attempt to integrate multiple governors to save more power while achieving a satisfactory performance, but have not considered memory traffic that can significantly affect performance and power consumption.

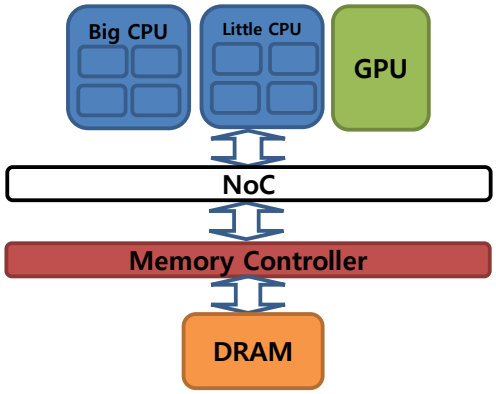


Figure 2.1: Recent heterogeneous architecture in mobile platform

Mobile games generally demand high memory throughput due to complicated graphics and high frame rate. Figure 2.2 shows the average memory throughput between NoC and the memory controller generated by different mobile applications. It clearly shows that the eight mobile game applications on the right demand higher memory throughput than the three general applications on the left. Since mobile games typically have complex graphics workloads and high frame rates, the memory is stressed due to large amount of memory requirement. Besides of mobile games, some mobile applications e.g. face recognition, can also uses GPU to accelerate as contemporary mobile GPUs usually support data parallel programs like OpenCL which might also lead to high utilization of memory bandwidth.

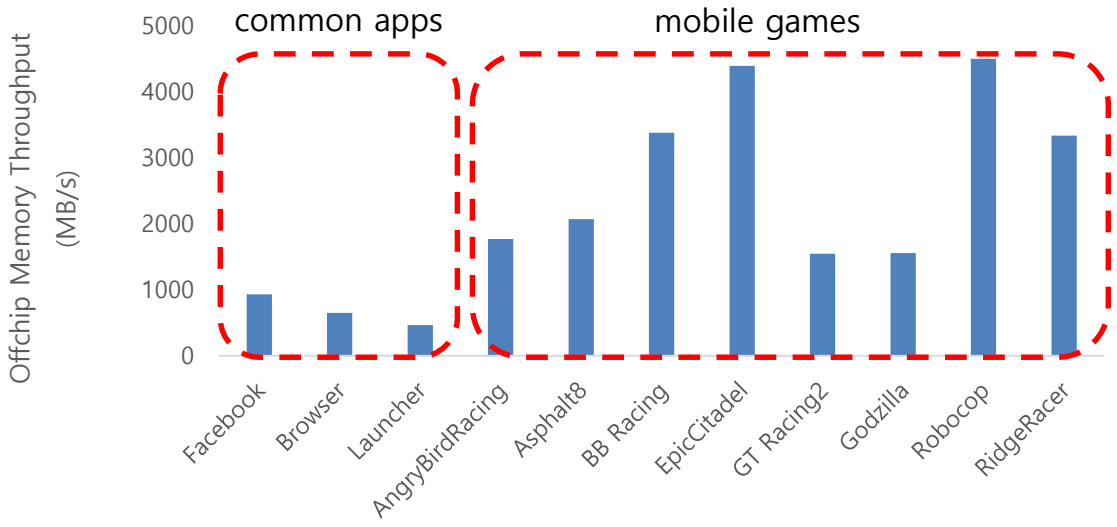


Figure 2.2: Memory throughput of mobile applications

Traditional heterogeneous CPU-GPU systems have CPU and GPU implemented on different chips with dedicated DRAM whereas mobile systems usually have a shared DRAM for both CPU and GPU cores as shown in Figure 2.1. Hence, the demand of high memory throughput from GPU can have a deeper impact of CPU performance than traditional architectures. Besides, the contemporary mobile systems have governors to perform memory and bus DVFS which further complicates the situation. Previous research [18] [19] [20] [21] has studied the influence of how CPU accesses the memory to making a CPU DVFS decision on both desktop and mobile systems. [22] starts to investigate the relation between performance and the priority of memory requests from CPU and GPU.

In this chapter, we present MEMCOP, a memory-aware runtime power and performance management framework that cooperatively orchestrates three governors: CPU, GPU and memory governors for mobile games in heterogeneous platforms. The MEMCOP cooperative game governor saves power and energy consumption using DVFS while still satisfying the performance goals.

The major contributions of this chapter are as follows:

- We perform characterization of off-chip memory usage for recent mobile games.

- We present a general regression model to predict the time spent in memory accesses which can be used for contemporary heterogeneous mobile architectures.
- We present MEMCOP, a memory-aware synergistic CPU-GPU DVFS governor for mobile games

To the best of our knowledge, this is the first work to coordinate CPU, GPU, and memory governors synergistically among mobile power management approaches for mobile games. With this model integrated into our DPM, we are able to further improve energy efficiency as well as the power consumption. Figure 2.3 shows a motivational example for the *Robocop* mobile game. We compared our MEMCOP cooperative governor with the default governor in perspectives of energy efficiency (Energy Per Frame), performance (Frame Per Second), and average power consumption. It demonstrates the MEMCOP cooperative governor can improve power and energy efficiency up to 20% with minor performance downgrade comparing with the default governors.

The remainder of this chapter is organized as follows. Section 2.2 describes background of mobile graphics and motivation. Section 2.3 presents related work. Section 2.4 introduces our MEMCOP power management structure including the memory-aware governor. Section 2.5 outlines the experimental results and shows the comparison between default, state-of-the-art, and MEMCOP governors. Finally, Section 2.6 concludes the work.

2.2 Related Work

There is a large body of research on saving power and energy consumption consumed by different components in mobile platforms. Figure 2.4 shows a general mobile power management structure. Many hardware components have separate governors to control performance and power consumption. The related work can be categorized into two groups: standalone

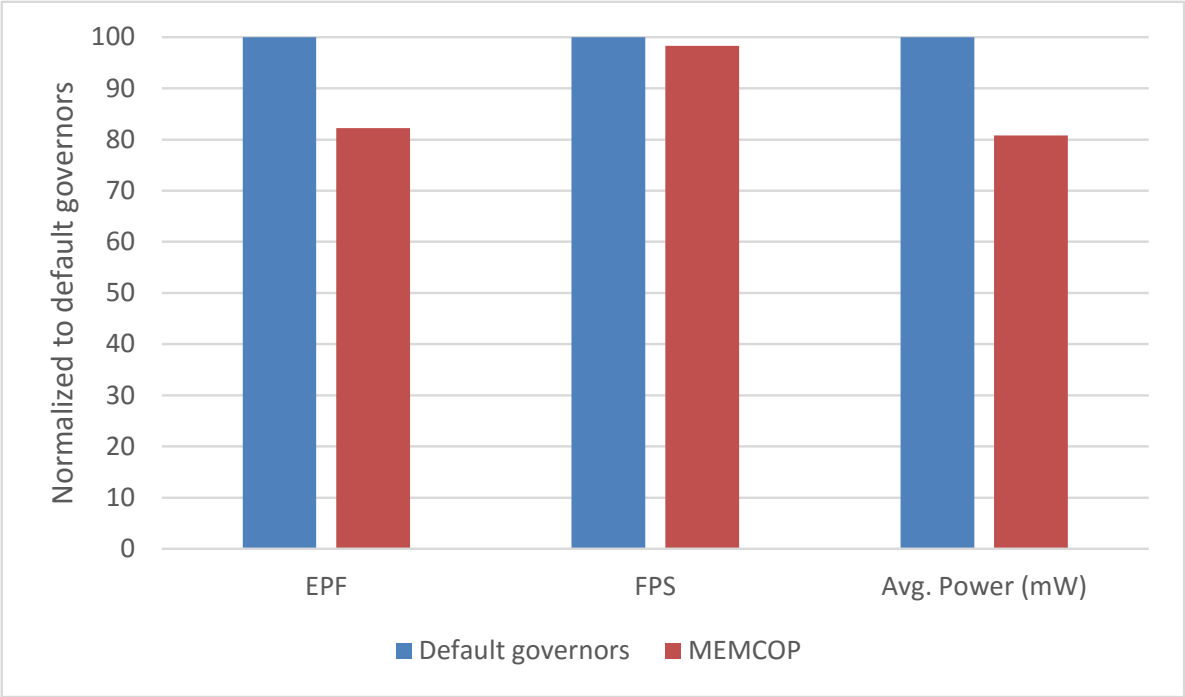


Figure 2.3: Comparison of default governor and MEMCOP memory-aware governor for the game: Robocop

governors addressing only one governor, and integrated governors that integrate two or more governors.

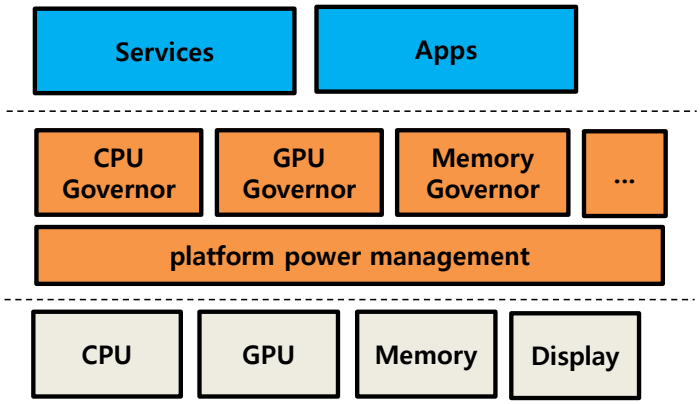


Figure 2.4: General mobile power management structure

2.2.1 Standalone Governors

CPU DVFS has been exploited to save power while playing mobile games. Gu et al. [23] proposed a CPU graphics rendering workload characterization and CPU DVFS for 3D games for mobile platforms where there is no integrated hardware graphics accelerator. More recent efforts have looked at contemporary mobile hardware architectures. Dietrich et al.[24] proposed a power management scheme which applies a linear model to predict the coming CPU workload based on previous graphics workload. Further, they built a game state detection mechanism and performed CPU DVFS to satisfy pre-defined QoS according to the current state to save power. Ercan et al.[25] proposed a heuristic to perform CPU DVFS by monitoring GPU utilization to reduce CPU power consumption with minimal performance downgrade. You et al.[26] proposed a heuristic approach to perform GPU DVFS to save power and meet the QoS requirement. Jeong et al. [22] showed that statically prioritized memory access requests from CPU causes poor performance of GPU and hence leads to unacceptably low frame rate. With a dynamic QoS policy changing the priority of CPU and GPU memory requests can improve frame rate significantly. Kim et al. [27] demonstrated a mechanism to calculate redundant frame rate and effective frame rate and to save power by dynamically changing the display refresh rate to eliminate redundant frame rate. All of the above efforts consider CPU, GPU, or memory separately to study their influence on performance and power consumption which limits their opportunity to achieve better energy efficiency through integrated governors when dealing with a large spectrum of mobile games.

2.2.2 Integrated Governors

Some recent efforts have proposed integrated governors considering the effects of both CPU and GPU. Pathania et al. [17] proposed an integrated power management for mobile games which coordinates both CPU and GPU. They showed that through cooperation between

CPU and GPU governor, the overall power can be reduced while playing mobile games. They also extend their work in [28] by using a sophisticated model which considers the interaction between devices' utilization, frequency, and the graphics performance. Our scheme is differentiated from previous work in a number of aspects. First, we present a MEMCOP, memory-aware integrated governor which coordinates CPU, GPU, and memory governors to minimize the energy consumption and still reaches performance goals. Second, we also use energy efficiency to evaluate different governors and show our MEMCOP governor outperforms the others. To the best of our knowledge, this is the first work to include memory traffic influence in integrated governors for mobile games.

2.3 MEMCOP: Memory-Aware Cooperative CPU-GPU DVFS Governor

2.3.1 Background

In hardware-accelerated mobile SoCs, graphics workload is usually handled by both CPU and GPU: the CPU works as the producer which generates rendering commands and the GPU consumes them. We choose Android platform which is one of the popular mobile operating systems to introduce some background for graphics rendering.

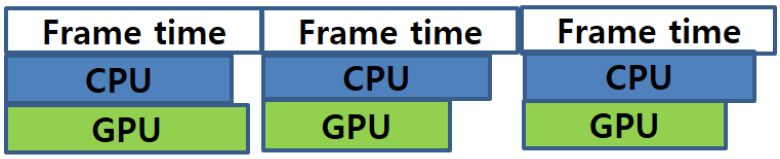


Figure 2.5: Abstract pipeline for graphics rendering

Graphics Rendering

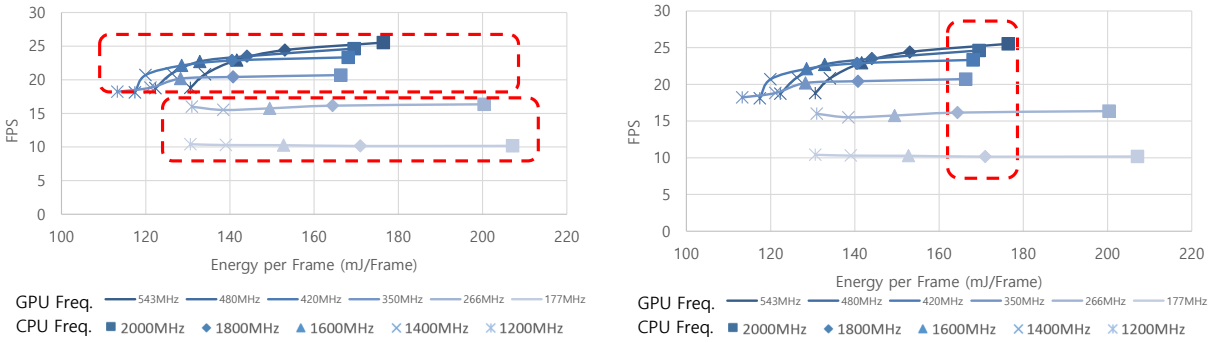
In the Android framework, graphics rendering is triggered by the vertical synchronization (VSYNC) signal which is generated by display, with multiple buffering applied instead of traditional double buffering to remove stuttering [29]. Therefore, CPU and GPU processing for graphics can be represented as a two-stage pipeline as shown in Figure 2.5. At higher level, a frame is composed of different layers which can be produced by different applications or system services. The system service SurfaceFlinger is activated by the VSYNC signal and starts to update a frame by scanning through the list of layers. Most mobile games use full screen mode, so a frame is usually generated by only the game process. The frequency at which SurfaceFlinger updates a frame is labeled Frame Per Second (FPS). FPS is a common metric to evaluate graphics performance and we also evaluate our work using FPS as a performance metric. As mentioned earlier, we found the memory bus is heavily utilized while playing mobile games and thus a large portion of execution time is spent in memory requests. Since the memory bus is a shared resource, time spent in memory requests depends on contention among the components competing each other to access the memory. One representative component is CPU and thus CPU memory access rate (MAR) should be carefully considered along with bus utilization in devising a model to predict the memory access time during graphics rendering.

Graphics Thread Model

An Android application usually creates tens of threads, with only a few being graphics related threads. Some of the graphics threads are created by userspace graphics libraries which are developed by a chip vendor to communicate with the graphics driver and whose source codes are not available. Our observations show that there is usually one game thread mainly affecting the graphics performance in most of the mobile games we tested.

For the convenience of our experiments, we profile each game and manually fix the highly graphics-related game thread on a dedicated core in all of our experiments.

2.3.2 Performance and Energy Efficiency for Mobile Games

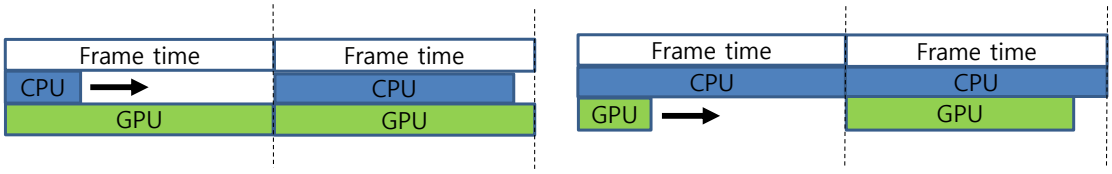


(a) Influence of CPU and GPU frequency scaling (b) Influence of choosing proper operating points

Figure 2.6: Average energy efficiency vs average FPS from running Antutu3D with different CPU/GPU frequencies

Our goal is to achieve optimal energy efficiency with minimal performance degradation. We use energy efficiency (energy per frame) as the primary metric to evaluate our work. Figure 2.6 shows the energy efficiency and FPS with different combinations of CPU and GPU frequencies with our experimental ARM-based platform introduced in Section 2.4.1. From Figure 2.6a, we can see as GPU frequency increases, the FPS improves significantly which indicates the graphics workload of the application is mostly GPU-bound. As GPU frequency goes up to certain level e.g. 350MHz, CPU frequency begins to affect the FPS as well. That implies for some of the frame workload, it is either CPU-bound or GPU-bound to reach the FPS goal. Figure 2.6b reveals that the cost of selecting an improper frequency can be huge. The difference of energy per frame between each CPU operating point increment significantly but the performance could still remain similar. The performance penalty of choosing a wrong frequency combination is also obvious which might lead to low QoS. In addition, it shows we can achieve a similar FPS with lower energy per frame (higher energy energy efficiency) by choosing appropriate energy-efficient frequency for CPU and GPU.

On the other hand, for similar energy efficiency, the FPS can be varied. In other words, high energy efficiency (low energy per frame) doesn't always mean the selected frequency combination is better, since the performance might be degraded heavily. Therefore, when we use energy per frame to evaluate different approaches, it's necessary to have a similar FPS performance for fair comparison.



(a) Reducing slack time on CPU execution (b) Reducing slack time on GPU execution

Figure 2.7: Reducing slack time in both CPU and GPU processing

The work proposed in [17], [30] and [24] have shown that as CPU and GPU have higher utilization during a period of time, the energy consumption can be reduced while maintaining FPS at a similar level. CPU and GPU utilization are represented by the ratio of busy cycles over total cycles which is usually provided by corresponding governors.

When we break down the rendering process to frame level, high utilization can be represented as low slack time during a frame. Figure 2.7 shows how to achieve high utilization at frame level granularity, which requires controlling the execution time of both CPU and GPU to match the frame time. Specifically, Figure 2.7a indicates the bottleneck for graphics rendering is on GPU, resulting in significant CPU slack time. Hence, reducing CPU slack time by adapting CPU frequency leads to lower power and energy consumption without downgrading FPS. As the energy consumption is reduced while remaining the same performance, the energy efficiency improves. On the other hand, similar idea can be applied on GPU when CPU becomes the bottleneck as shown in Figure 2.7b. In our work, we use a frame-based approach to measure the execution time of CPU and GPU workload periodically and reduce the slack time as much as possible by using DVFS.

2.3.3 Memory Access Behavior of Mobile Games

Terminology

Before going into the details of our MEMCOP power management structure for mobile games, we start with definitions of some terms used to describe the memory access behavior.

- **Memory Utilization** (MU) is the percentage of memory traffic used out of maximum bandwidth of the shared bus between NoC and memory controller. The value represents the degree at which the system stresses the memory interface.

- **Memory Access Rate** (MAR) has been adopted in many memory-aware algorithms [18]. We are mainly interested in the off-chip memory accesses, so we defined MAR as the ratio of number of last level data cache misses and number of instruction executed. The formula for MAR is:

$$MAR = N_{llc\_miss}/N_{inst\_exec} \tag{2.1}$$

  We use CPU MAR to describe the level at which CPU workload accesses off-chip memory.

- **Memory Access Time** is defined as the total time spent after memory requests are issued and before receiving responses from off-chip memory during a frame. Hence, it is independent of CPU or GPU frequency. It is expected that memory access time is dependent on memory frequency. Also, memory DVFS can lead to higher power and energy consumption which makes it unattractive for power management schemes [17]. Therefore, we fix memory frequency at the highest frequency level.

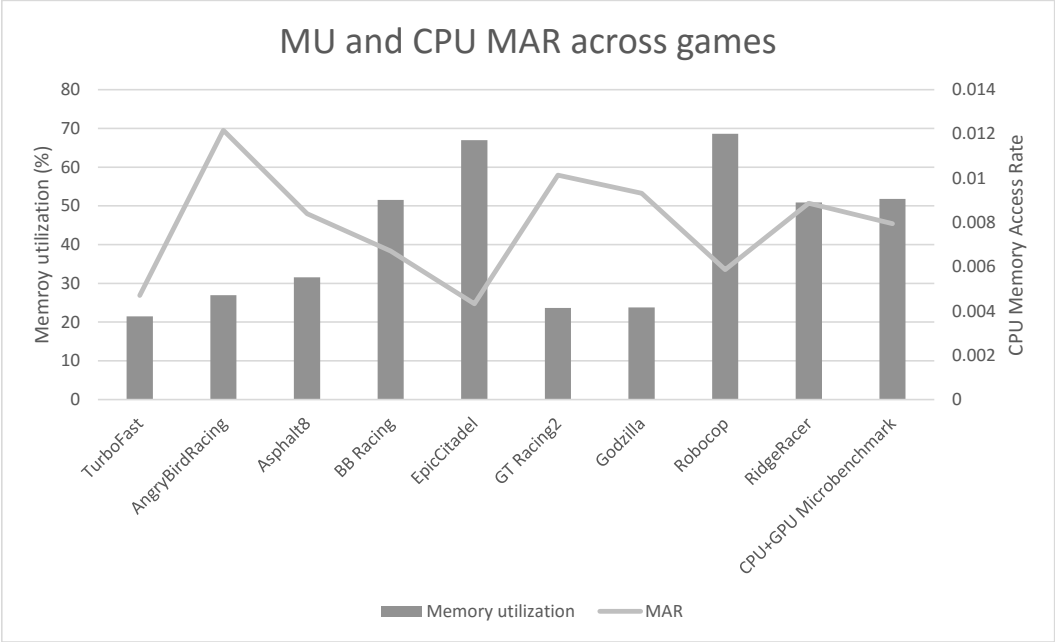


Figure 2.8: MU and CPU MAR inter games

MU and CPU MAR across mobile games

We first investigate the memory access behavior in different mobile games. We characterize memory access behavior with MU and CPU MAR which represent how the system stresses memory interface and how CPU workload of games access memory respectively. As Figure 2.8 shows, the average MU and MAR of mobile games are distinct from each other. The MU variation comes from the fact that mobile games have different complexity of graphics such as the resolution of image quality and the game scenes which leads to different memory requirement from GPU. The CPU MAR variation can be attributed to the variation in the complexity of the game logics.

MU and CPU MAR intra mobile games

We measure the variation of MU and MAR within each game frame by frame. Figure 2.9 shows an example of how they vary within a mobile game (BB Racing). As the game is

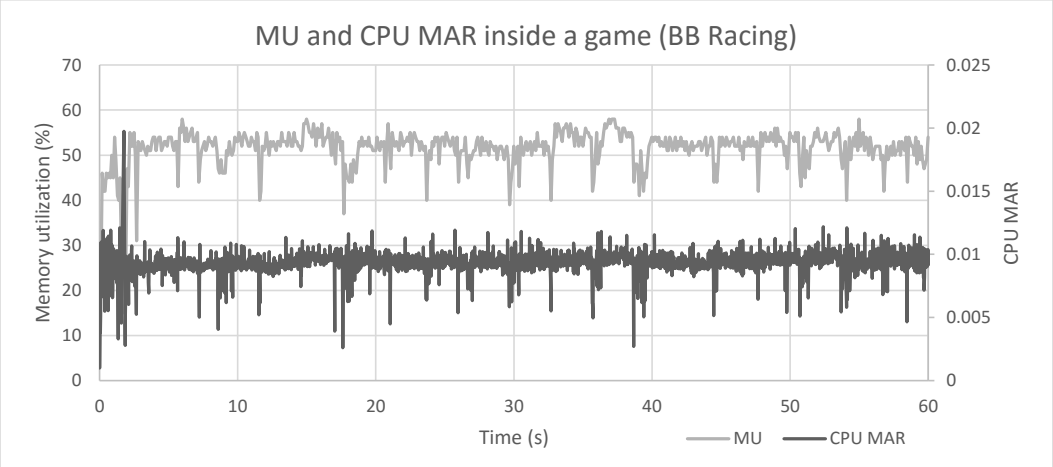


Figure 2.9: MU and CPU MAR intra games

running, the CPU and GPU workload are changing along with the time and the user inputs. Therefore, the memory access behavior of CPU and GPU vary dynamically.

MU, CPU MAR and CPU execution time

Here, we are interested in how much of CPU execution time is spent in accessing memory so that we can apply the information to our DVFS algorithm. Figure 2.10 plots the results of running our custom memory-intensive CPU benchmark which is set to have a regular CPU MAR together with a mobile GPU benchmark used to change MU. The main observation is that the execution time is changing along with memory utilization. Since CPU execution time are also affected by MAR natively, this motivates us to build a correlated equation to predict memory access time by CPU MAR and MU.

2.3.4 Memory-aware Cooperative CPU-GPU DVFS governor

Figure 2.11 presents the overview of our MEMCOP governor built on top of the CPU, GPU and memory governors and which communicates with them periodically. The MEMCOP cooperative governor makes a DVFS decision when a new frame is pushed out to display. The

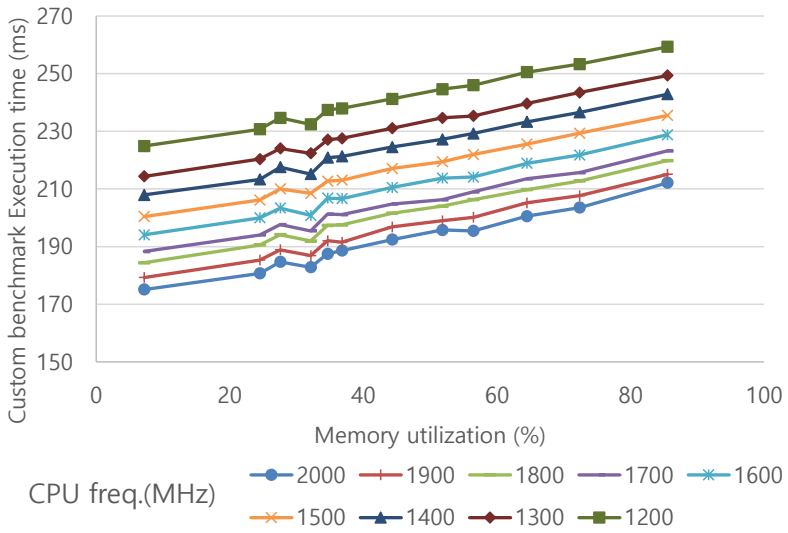


Figure 2.10: Memory utilization and execution time

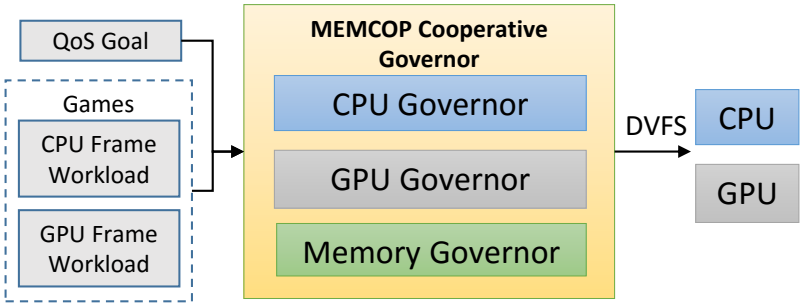


Figure 2.11: Overview of MEMCOP memory-aware cooperative CPU-GPU Governor

underlying three governors are also executed synchronously to measure their performance. Hence, the measured data represent the results from last frame workload. Here we assume the workload of coming frame will be similar to previous one and use the measured data to predict the performance of coming frame after making a DVFS decision for the CPU and GPU. The assumption is built on the fact that game graphics workloads are usually generated at runtime. Consequently, the workload changes over time as well as user input according to the game logic. As long as the game scene doesn't change, most of the objects rendered remain similar. Hence, we assume the frame workload is similar to the previous one. MEMCOP governor is also capable of receiving a QoS goal from the application level which is the goal to perform DVFS. The QoS goal is set to maximum possible FPS by default.

Table 2.1: Parameters used in our prediction model

Name	Description
ft_{goal}	The goal frame time
ft_{min}	Min frame time that is the reciprocal of the max screen refresh rate
ft	Frame time
ft_c	CPU frame time
ft_g	GPU frame time
F_c	CPU frequency
F_g	GPU frequency
T_s	Part of ft_cscaled by CPU frequency
T_{ns}	Non-scalable part of ft_cbut affected by MAR and MU
T_b	The difference between real and goal frame time
MAR	The ratio of number of last level data cache misses and number of instruction executed
MU	The percentage of memory traffic used out of maximum bandwidth of the shared bus

Cooperative Governor Algorithm

Here we introduce the models we adopted and how to achieve the Qos goal by using them. Table 2.1 lists all the used parameters. The parameters reflect the measurement result of the latest frame processed. The parameters attached with a prime symbol (') indicate that they are for the coming frame workload. Specifically, our goal is to make ft as close as ft_{goal}. The cooperative governor is invoked every frame. As described previously, the energy is minimized by fully utilizing the CPU and GPU. Hence, the governor performs CPU and GPU DVFS to control the execution time to be close to the QoS goal. We first convert QoS goal into a frame time goal:

$$ft_{goal} = 1/fps_{goal}$$

where ft_{goal} is the performance goal for both CPU and GPU while handling frame workload. The CPU and GPU execution time of a frame are defined as ft_c and ft_g respectively. For a

given frame workload with ft_c and ft_g, we can express the frame time as:

$$ft = \begin{cases} max(ft_c, ft_g) & \\ ft_{min} & \text{if } ft_c, ft_g < ft_{min} \end{cases} \tag{2.2}$$

For the cases where CPU or GPU is saturated and cannot reach the QoS goal, the bottleneck will become the new QoS goal for predicting the frequency for next frames since there is room for the non-bottleneck unit to save energy. In Equation (2.2), ft_{min} represents the minimum possible frame time since the maximum FPS is limited by hardware usually configured as 60 FPS in Android systems. In practice, we are not able to match ft_c and ft_g with ft_{goal} due to the following reasons: 1. There is no operating point in the platform for the predicted frequency, and 2. Although we assume the graphics workload remains the same as the previous one, the workload might change slightly in practice which leads to variation of frame time. Therefore, the frame time above or below ft_{goal} should be considered to achieve QoS requirement with minimum energy consumption. We define T_b as the difference between the real frame time and goal frame time. The QoS goal can modeled as:

$$ft'_{goal} = ft_{goal} + T_b \tag{2.3}$$

ft_{goal}' represents the new goal and the time required by the bottleneck device. For mobile graphics rendering, it is not always feasible to compensate the loss of FPS. Due to hardware limitation, if the QoS goal is set as ft_{min}, there is no way to compensate the extra time spent in rendering since the rendering happens with a minimum interval ft_{min}. Another situation is when the workload saturates CPU or GPU. In other words, the workload requires maximum frequencies of CPU or GPU to meet the QoS requirement. Thus, it's also impossible to level off the extra rendering time since there is no higher frequency settings. In both cases, T_b can be ignored. In some cases e.g. scene change, the workload varies substantially which leads to a huge T_b. Hence, we reset the ft_{goal} when a new ft_{goal} is half of ft_{goal} larger or smaller

than the original goal.

For ft_g prediction of the GPU, we adopt a scaling model similar to the performance-cost model used by Pathania et al. [17]:

$$ft'_g = ft_g * F_g/F'_g \tag{2.4}$$

where F_g and F'_g is the GPU frequency used for the latest and the coming frame respectively and ft'_g is the predicted GPU execution time for the coming frame. For prediction of CPU, we express ft_c with frequency scalable time T_s and non-scalable time T_{ns} [31]. T_s can be computed by a scaling model similar to Equation (2.4). T_{ns} is characterized as the time to access off-chip memory hence it is non-scalable with respect to CPU frequency.

$$\begin{aligned} ft'_c &= T'_s + T_{ns} \\ T'_s &= T_s * F_c/F'_c \end{aligned} \tag{2.5}$$

Similarly, F_c and F'_c is the CPU frequency used for latest and the coming frame respectively. T'_s and ft'_c is the predicted scalable part of execution time and predicted total CPU execution time for the coming frame respectively. The performance goal can be modeled as:

$$\begin{aligned} ft_{goal} &\geq ft'_c \\ ft_{goal} &\geq ft'_g \end{aligned} \tag{2.6}$$

Therefore, we can derive the equations for predicting the frequencies of CPU and GPU for the coming frame from Equations (2.4), (2.5) and (2.6):

$$\begin{aligned} F'_g &\geq ft_g * F_g/ft_{goal} \\ F'_c &\geq T_s * F_c/(ft_{goal} - T_{ns}) \end{aligned} \tag{2.7}$$

From Equation (2.7), the operational frequencies for CPU and GPU can be generated. Since

these exact frequencies might not exist, we select the lowest frequencies higher than the predicted ones to achieve the performance goal. The extra time saved because of using higher frequencies can be amortized by Equation (2.3) in the coming frames.

CPU Memory Access Time Prediction

It's usually infeasible to retrieve memory access time directly from hardware counters since they are typically confidential in commercial platforms. Hence, we construct a regression model to estimate the memory access time which is presented as T_{ns} in Equation (2.5). As we described in Section 2.3, CPU memory access time is highly impacted by MAR and MU, thus T_{ns} can be modeled as:

$$\begin{aligned} T_{ns} &= ft * \%T_{ns} \\ \%T_{ns} &= c1 * MAR + c2 * MU + c3, \end{aligned} \tag{2.8}$$

where $\%T_{ns}$ presents the percentage of a frame time spent in accessing off-chip memory. MAR is calculated from CPU hardware performance counter. MU can be retrieved from the memory governor. We built a custom memory benchmark to generate different levels of MAR. MU is controlled by a graphics benchmark [32] in which we used maximum texture and frame buffer size and then varied the number of triangles rendered to increase the memory utilization.

In summary, the governor performs CPU/GPU DVFS and predicts the CPU/GPU frame time individually according to the measurement results from the latest frame workload. As the governor achieves to adjust CPU/GPU frame time as close as the goal frame time, the users won't experience performance downgrade and the energy can be saved which leads to better energy efficiency in all.

Table 2.2: Platform Configuration

Feature	Description
Device	Odroid-XU3
SoC	Samsung Exynos 5422
CPU	ARM Cortex-A15 and Cortex-A7 big.LITTLE, 2.2Ghz
GPU	Mali-T628 MP6, 543Mhz
System RAM	2GB RAM (933Mhz Dual-ch.)
Mem. Bandwidth	up to 14.9GB/sec
OS(Platform)	Android 4.4.2
Linux Kernel	3.10.9

2.4 Experiments

2.4.1 Experimental Setup and Methodology

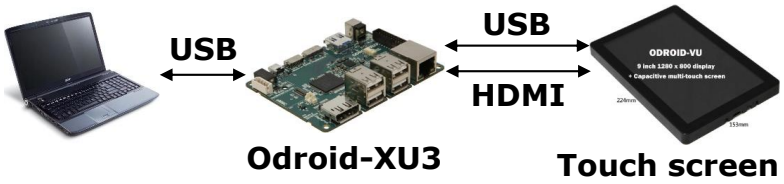


Figure 2.12: Experimental setup

Table 2.2 summarizes our hardware and software platform configurations used in our experiments. We used Odroid-XU3 [33] development board installed with Android 4.4.2 and Linux 3.10.9. As shown in Figure 2.12, the board is connected with a touch screen and a host PC where we collect the experimental data and profiling results. The board is equipped with four TI INA231 power sensors measuring the power consumption of big CPU cluster, little CPU cluster, GPU and memory respectively. We run all the benchmarks and games on big CPU cluster since we focus on the interactions between CPU, GPU and memory bus traffic. The big CPU cluster supports cluster-based DVFS at nine frequency levels by default. The performance measurement units are enabled for the governor to query the results at runtime. The GPU supports operation at six frequency levels and the execution time needed by our governor is provided by GPU governor. The memory utilization is provided by memory

governor which accesses performance counters on NoC. The power data points are collected every 125ms by creating a Linux kernel thread.

Table 2.3 summarizes the benchmarks we used for model building. We use GPU micro-benchmarks [30] to analyze correlation of the performance and energy efficiency on the GPU. We use GPU Performance Analyzer [32] and our synthetic memory benchmark to control the MU and MAR respectively for building the regression model to predict CPU execution time. We use LMBench to approximate the off-chip average memory access latency used in a memory-aware naive governor. The governor will be described in detail in later section. Table 2.4 shows characterization of the tested games from the perspectives of CPU, GPU and memory utilization when running with default governors where we attempted to run games exercising all the combinations of CPU, GPU, and memory intensity. For instance, in the extreme example, *Robocop* runs at higher CPU and GPU frequencies with high memory throughput during the experiments.

Table 2.3: Experimental benchmarks

Benchmark	Description
GPU micro-benchmarks [30]	GPU energy efficiency analyses
GPU Performance Analyzer [32]	Adjust memory utilization for model building
Custom memory-intensive benchmark	Adjust CPU MAR for model building
LMBench benchmark [34]	Approximate average memory access latency

Table 2.4: Characterization of experimental games

Games	CPU	GPU	Memory
Turbo Fast	High	Low	Low
AngryBirdRacing	Low	Low	Low
Asphalt8	High	Low	Low
RidgeRacer	Low	Low	High
BB Racing	Low	Low	High
GT Racing2	High	Low	Low
EpicCitadel	Low	High	High
Godzilla	Low	Low	Low
Robocop	High	High	High

Implementation In general, the default governors reside in kernel space and the userspace programs can configure or retrieve the status of the devices through some interfaces e.g. sysfs filesystem which is a feature of Linux enabling the communication between userspace and kernel space programs. The default governors work independently and periodically with possibly different intervals. They also make DVFS decisions based on their own measurement results only. Instead, our MEMCOP governor synchronizes the three governors: CPU, GPU and memory governor and make a DVFS decision considering all the information from the three components. Our governor is divided into user space and kernel space as shown in Figure 2.13. The Android service SurfaceFlinger is modified to send a notification to the cooperative governor through sysfs after it receives the VSYNC signal from display. Userspace applications can set up the QoS goal through the same interface. The cooperative governor sits in kernel space. We first synchronize CPU, GPU and memory governors. CPU and memory governors are configured to use userspace governor which does nothing when they are waked up periodically. GPU governor is modified to perform only measurement. When MEMCOP governor receives the notification from SurfaceFlinger, it collects measurement results from the other governors. We use performance counters to measure CPU. As there is no documentation describing how to access the hardware performance counters of GPU and memory subsystem, we simply collect the GPU and memory utilization statistics from the original governors. Then MEMCOP performs the DVFS algorithm with those results.

2.4.2 Not All F-V Settings Are Useful

In order to analyze the relationship of FPS and energy efficiency on our experimental platform, we ran GPU micro-benchmarks that exercise different components of the GPU [30]. Figure 2.14a shows the FPS results of running vertex memory micro-benchmark (mbVerM) at different GPU frequency settings where workload factor presents the extent of stressing a specific component in GPU pipeline (e.g. the amount vertex memory read). The results

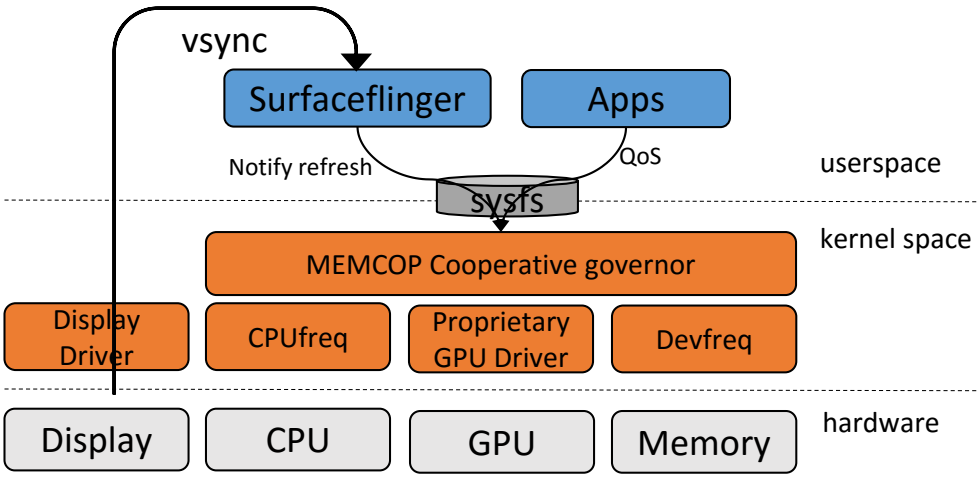
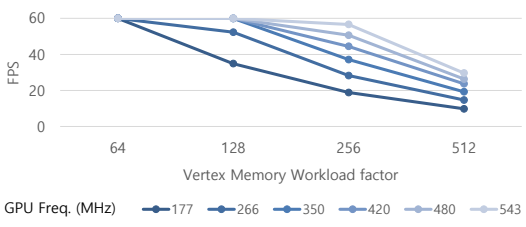
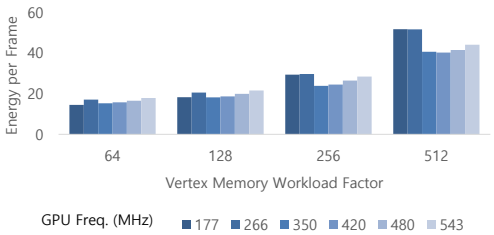


Figure 2.13: MEMCOP implementation overview

indicate FPS varies along with GPU frequency and while GPU is saturated, but decreases as workload increases by using more vertex memory. However, as we can see from Figure 2.14b, the energy efficiency of 350MHz outperforms that of 266MHz in every benchmark configuration. The same result is observed in other micro-benchmarks. Hence, we specifically skip using this frequency in our work. We also speculate that the frequency-voltage setting of 266MHz is not fine-tuned for this platform. As Figure 2.14b shows, the energy per frame increases along with the frequency when workload factor is 64 and 128. However, the energy per frame of 266MHz is very close to 543MHz's. Therefore, according to this trend the energy per frame should be in between the bars of 177MHz and 350MHz. The same situation is also observed in other micro-benchmarks. Therefore, we speculate the configuration of 266MHz is not fine-tuned.



(a) Frame rate of vertex memory micro-benchmark



(b) Energy efficiency of vertex memory micro-benchmark

Figure 2.14: Results of vertex memory microbenchmark

2.4.3 Results and Analysis

We compare our MEMCOP cooperative governor against default, memory-unaware state-of-the-art, and naive memory-aware governors as shown in Table 2.5. Default governors represent independent CPU and GPU governors in Linux. It runs *interactive* and *default* governors for CPU and GPU respectively. State-of-the-art governor is represented as *PAT15* [28] is memory-unaware governor which is the follow-up work of [17]. It uses a more sophisticated model than its previous work to predict FPS according to current frequencies and utilization of CPU and GPU. It also requires a 3-second initialization to profile the current game scene by running CPU or GPU at highest frequencies and apply the results for later DVFS decisions. We exclude this initialization of PAT15 as it consumes lots of power and fairly focus on the performance of the DVFS algorithms. The naive memory-aware governor is represented as *NaiveMem* in the figures which assumes the off-chip average memory access latency is fixed without the influence of MU and MAR. Then the memory access time is calculated as the product of the number of last level cache misses and average memory access latency. The latency is approximated by LMbench benchmark suite [34]. Each game is played for three times and each runs for one minute. Figure 2.15 shows the performance comparison between these governors. We observe that our MEMCOP governor has only a minor 1.7% FPS degradation compared to the default governor the *PAT15* governor.

Table 2.5: Brief of game governors for experiments

Governor	Description
Default	Interactive governor and proprietary GPU governor (Default)
PAT15 [28]	Memory-unaware governor which develops a regression-based model to predict frequencies for CPU and GPU
NaiveMem	Memory-aware governor which runs under the assumption that memory access time is the product of number of main memory accesses and memory access latency
MEMCOP	Our cooperative memory-aware governor

The average total power consumption of different governors is depicted in Figure 2.16. Our

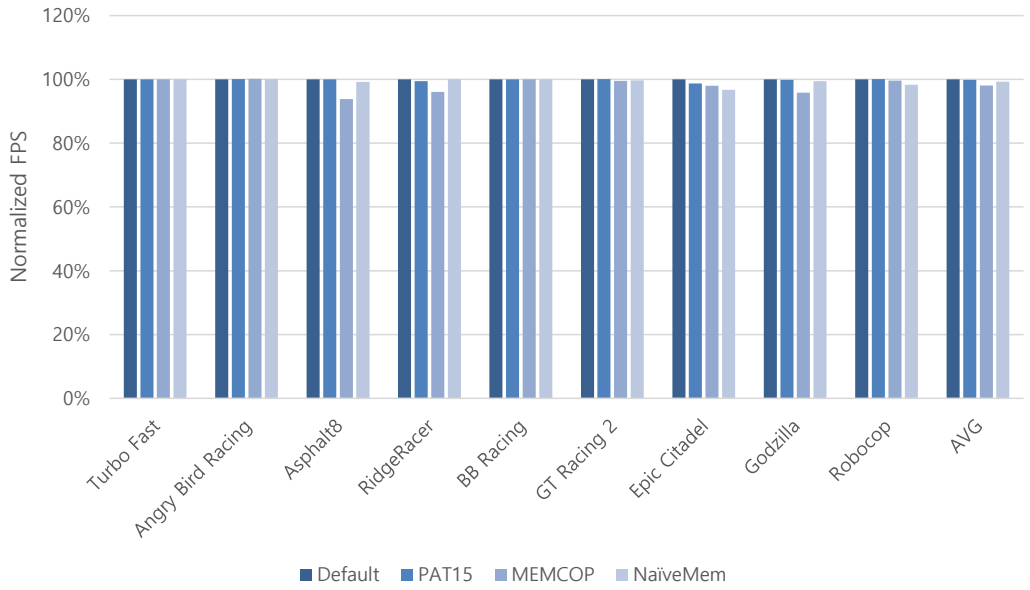


Figure 2.15: Frame rate between different governors

MEMCOP governor outperforms default governor by 22% less power consumption and 10% w.r.t. *PAT15* on average. It is obvious to see *NaiveMem* has a conspicuous high power consumption due to its naive assumptions so that it always predicts a high CPU frequency for the coming frame workload.

As Figure 2.17 shows, compared to default governor, our MEMCOP governor can reach up to 54% energy efficiency improvement without performance downgrade and 18% on average. Compared to *PAT15*, up to 20% energy efficiency improvement and 9% on average can be acquired without compromising the performance. For games with high memory utilization like *Epic Citadel*, we don't observe a great improvement, since the frame workload mainly stresses GPU, the CPU already runs at lower frequencies. Hence, there is not much to improve in these games. We also observe for some games like *GT Racing 2* and *Turbo Fast*, they create background threads which are not related to graphics performance. In this case, the default governor tends to use higher CPU frequency than the one that can meet the QoS requirement because those non-graphics related threads need more computing power than graphics rendering threads. Therefore, MEMCOP power management structure is able to improve energy efficiency significantly due to the discovery of graphics related threads

and memory-aware algorithm. For games like Epic Citadel and Godzilla, PAT15 has worse energy efficiency than the default governor. We speculate that because these games have high variation of game workload in short period of time, the initial profiling results are not capable of providing useful information for PAT15's algorithm to make energy efficient decisions.

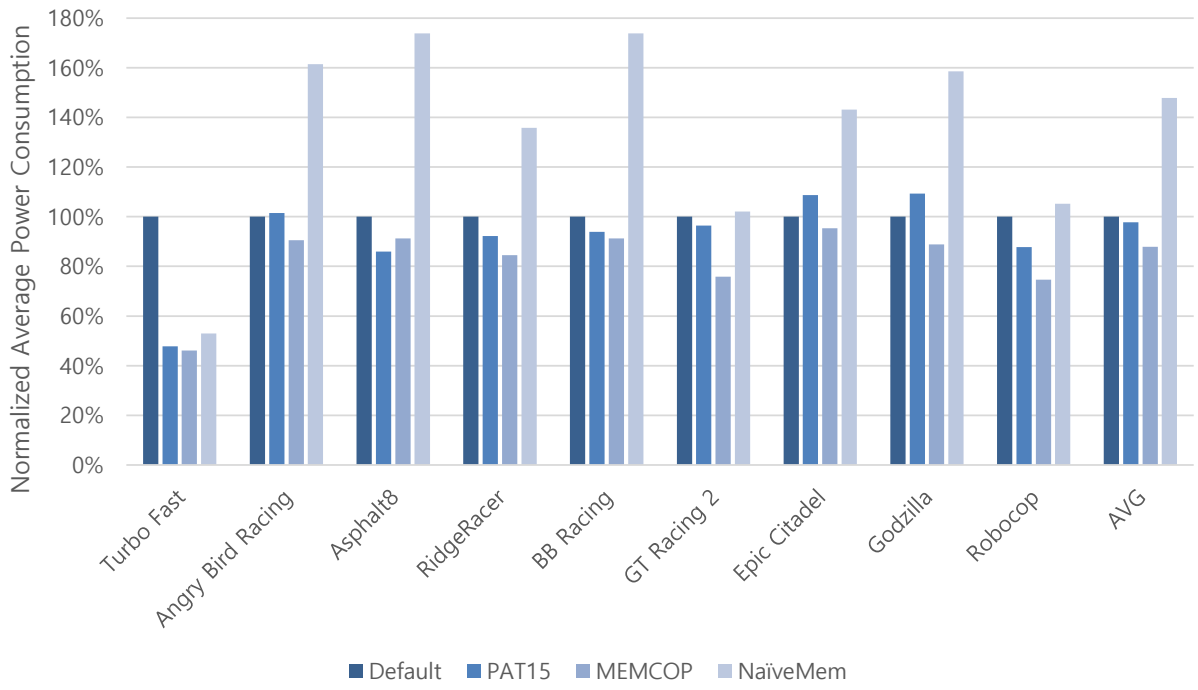


Figure 2.16: Total power consumption between different governors

MEMCOP adopts the idea to improve energy efficiency by reducing the slack time explained in Section 2.3.2 so we also perform slack time measurements. Figure 2.18 provides the normalized CPU and GPU slack time while running the applications. MEMCOP reduces slack time by around 8% and 42% on average CPU and GPU execution time respectively compared to default governor. The default governor and PAT15 have different epoch of running their algorithms, so we accumulate the busy and slack time of CPU and GPU for each epoch and calculate the percentage of slack time versus entire running time. In general, the slack time reflects the trend of how the energy efficiency is improved. However, in some of the games e.g. GT Racing 2, the trend is not obvious. As we mentioned in

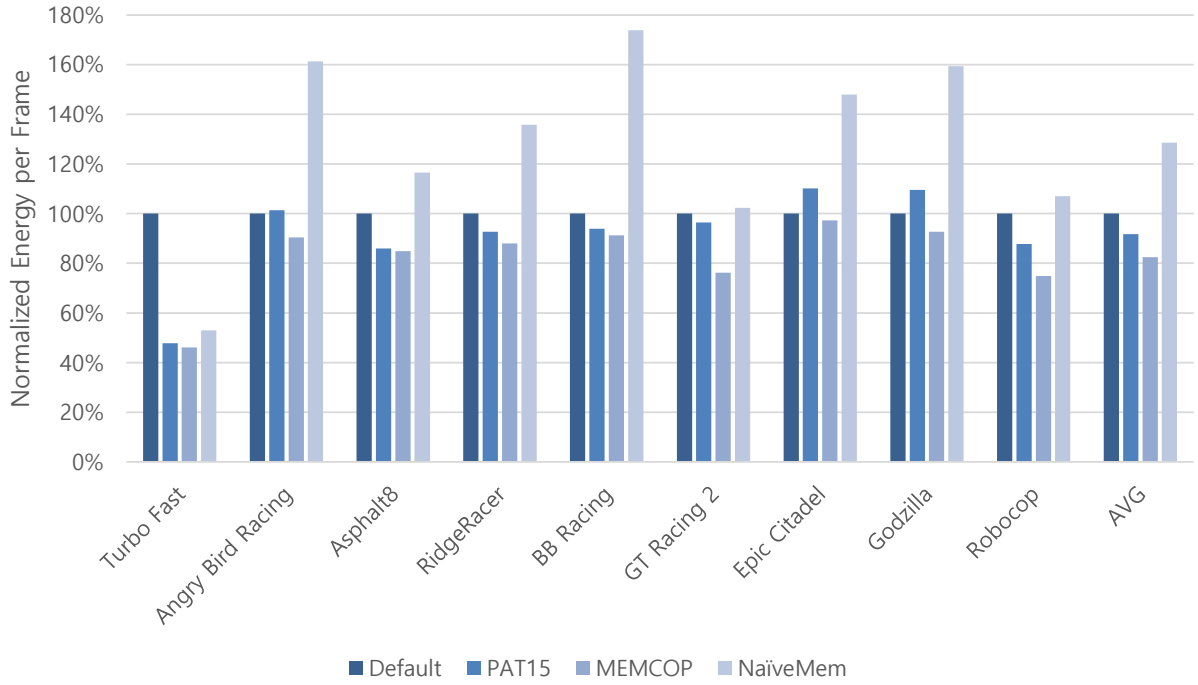
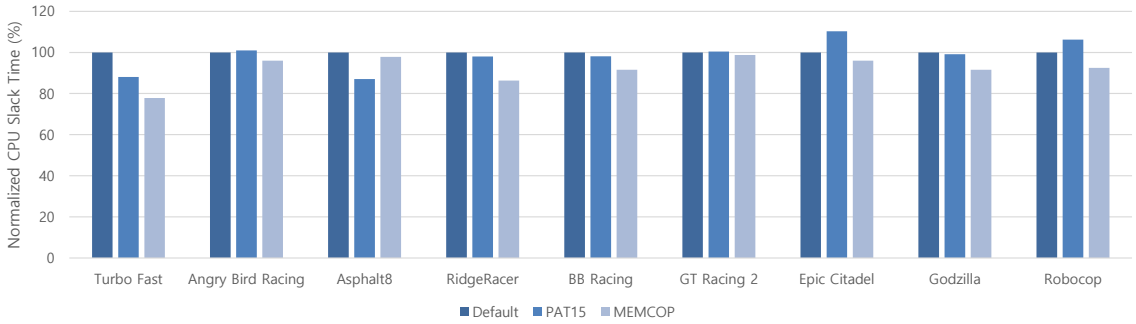


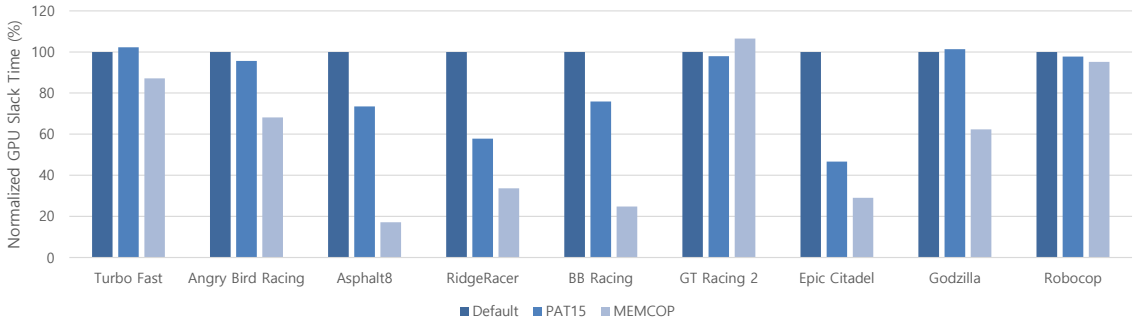
Figure 2.17: Energy per frame between different governors

previous paragraph, some of the games created background processes to perform intensive CPU computation task. The graphics rendering task is relatively ignorable. Hence, the use of high CPU frequency is not necessary for graphics rendering which is considered in our MEMCOP governor. In the results, the reduced slack time is significant for GPU in MEMCOP because the GPU operable frequencies have around 13 - 50% difference in each operating points so that the decision of scaling to what frequency results in obvious execution time difference. Instead, CPU frequencies have low difference in each frequency step (5 - 9%), therefore the reduced slack time is not obvious.

In conclusion, the results show our MEMCOP governor is able to save significant power and energy consumption with minor performance degradation compared to the default governor. Compared to memory-unaware state-of-the-art governor, ours is also able to reduce power and energy with a stable performance while off-chip memory is heavily stressed.



(a) Normalized slack time of CPU



(b) Normalized slack time of GPU

Figure 2.18: Normalized slack time of CPU and GPU from running different governors

2.4.4 Model Evaluation

We build the regression model by running two benchmarks: GPU performance analyzer [32] and our custom memory benchmark which are capable of changing MU and MAR respectively. For instance, we can change MU by configuring the number of triangles rendered in GPU performance analyzer. There are 12 levels of MU ranging from 7% to 85% in our tests which cover the MU generated by most of the games according to our observation. In our custom memory benchmark, it randomly accesses a given size of array for given number of times to generate different MAR. The range of MAR varies from 10^{-5} to 0.014. We dedicate one of the big cores to run the memory benchmark with GPU performance analyzer running in the other cores. CPU performance measurement units are activated to record number of last level cache misses, number of instructions executed and active cycles. By alternating CPU frequency, the non-scalable execution time can be calculated by Equation (2.5). Thereby the regression model is constructed by repeating this experiment under different

benchmark configurations and CPU frequencies.

From the perspective of performance in FPS, the MEMCOP governor can achieve an average 0.5% prediction error rate. As our approach predicts both CPU and GPU frame time by selecting an energy efficient frequency, we are more interested in how the prediction works. Figure 2.19 shows the average prediction error of both CPU and GPU frame time in our model which are 15.51% and 13.54% respectively. The prediction error is mainly from the dynamic

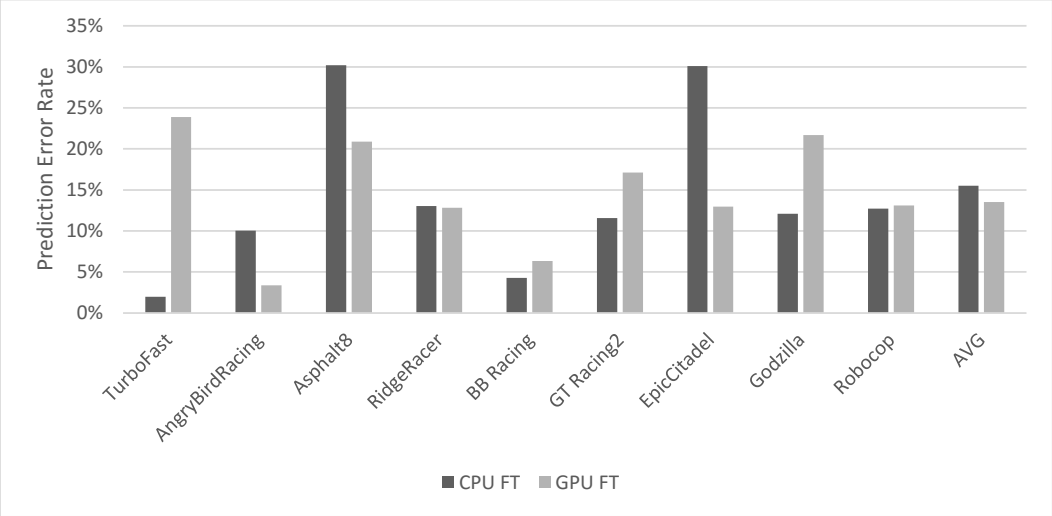
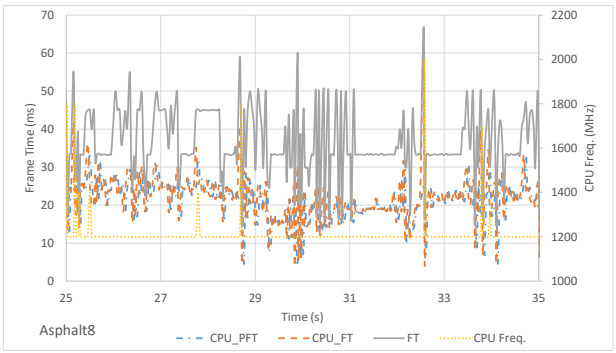


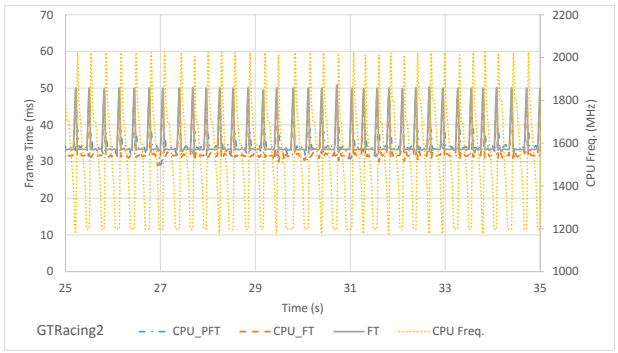
Figure 2.19: Prediction error for CPU and GPU frame time

workload generated by games. Since our governor makes decisions based on the assumption that the workload of the coming frame will be equal to the previous frame, the changes between each frame leads to the prediction error. Figure 2.20 shows an excerpt of runtime profiling results of predicted CPU frame time (CPU_PFT), CPU frame time (CPU_FT), frame time (FT), and CPU frequency for four games: *Asphalt8, GT Racing 2, Ridge Racer, and BB Racing*. The results also reveal different characteristics in these games. *Asphalt8* shows a high dynamic workload which leads to higher prediction error rate. The performance is mainly affected by GPU frame time since CPU frame time is mostly lower than total frame time which indicates that GPU is the bottleneck. *GT Racing 2* has regular spikes in CPU frame time which means the game thread periodically processes heavier workload and that also affects the performance using MEMCOP governor. As the default governor always

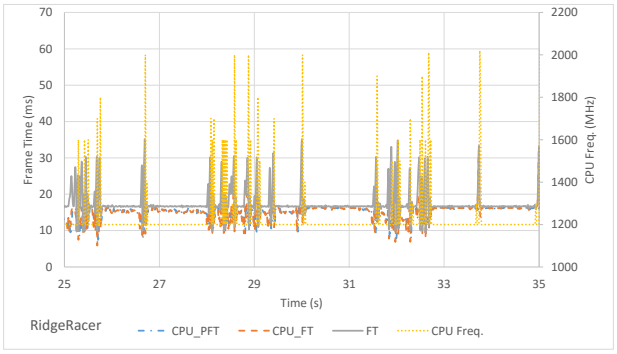
runs at higher frequency for this game, the MEMCOP governor has performance downgrade comparatively. *Ridge Racer* has less variation in CPU workload but the variation becomes bottleneck which affects performance. *BB Racing* also has less variation and most of them can be done within the frame time. However, the error doesn't deteriorate the performance significantly in the MEMCOP governor for two reasons: 1) Non-bottleneck execution: as we mentioned in Equation (2.2), the bottleneck of performance can be any one of CPU, GPU or none of them. In the case of non-bottleneck CPU or GPU execution, the variation of execution time will not impact the performance. 2) Performance-aware decision making: the exact number of predicted frequency is usually not one of the operating points provided by the system. For preventing downgrading of performance, the governor has to choose the minimum frequency higher than the predicted one. Due to the coarse-grained operating points, the selected frequency is usually much higher than the predicted one. Therefore, the governor can tolerate the variation of workload to some extent. The control overhead of



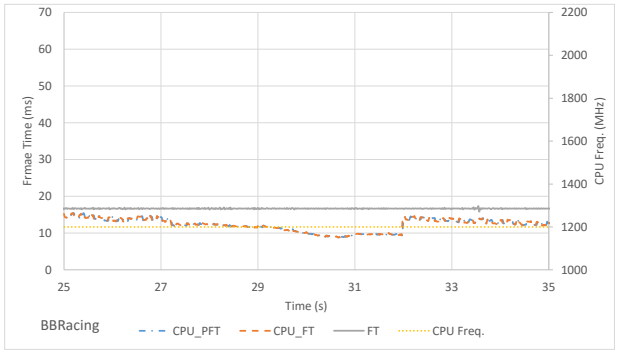
(a) Asphalt8



(b) GTRacing2



(c) RidgeRacer



(d) BBRacing

Figure 2.20: Profiling results when running the games with MEMCOP governor

the MEMCOP governor is less than 1% which doesn't include the time performing DVFS. However, CPU and GPU frequency scaling are cumbersome which both take around 1ms to complete. That accounts up to 12% overhead when the games are running at 60 FPS. Hence, it's possible the frame-based approach might further bring down the performance of games which saturate CPU at highest frequency most of the time or it has to operate at higher frequency to amortize the overhead. We also consider about the DVFS overhead in our MEMCOP governor.

2.5 Discussion

While our MEMCOP governor has outperformed the state-of-the-art governors, there are still many promising and unexplored points by which the energy efficiency could be further improved.

2.5.1 Workload Prediction

CPU Workload prediction for games has been researched for many years [23][35][36]. As we mentioned in evaluation section, the dynamic workload generated by the games causes imprecise frame time prediction if we assume the coming frame workload is the same as the latest one. This influence to performance might be mitigated by developing a workload prediction algorithm before performing DVFS algorithm. As the graphics performance is intertwined with CPU and GPU computation, we also need GPU workload predictor to increase the accuracy of GPU frame time prediction. Besides, if the workload predictor determines the workload of following frames is similar to the latest one, the governor doesn't have to perform DVFS frame by frame so that the DVFS overhead can be reduced as well.

2.5.2 Game Thread Scheduler

As we mentioned in Section 2.3.1, mobile games usually have one thread mainly affecting the graphics performance. Figure 2.21 shows how the performance variation when we explicitly fix the game thread we observed running at little or big cores which both operate at fixed frequencies. It's necessary for some of the games to run on the big cores to acquire full performance. From an energy efficiency perspective, the scheduler can perform game thread migration between little and big cores to potentially reduce energy consumption without performance downgrade. We also observe that some benchmarks e.g. *GPUBench* creates multiple graphics worker threads. Therefore, a game scheduler could potentially increase the energy efficiency by appropriately allocating resources for these graphics related threads.

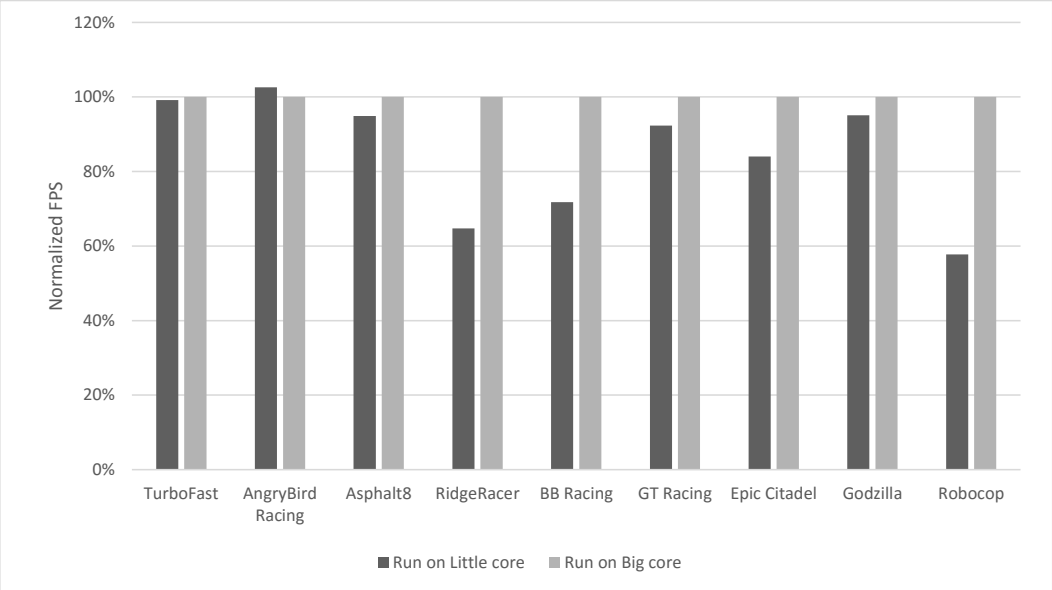


Figure 2.21: Performance profiling when the game thread runs at little or big cores

2.6 Conclusion

In this paper, we presented MEMCOP, a memory-aware cooperative CPU-GPU DVFS governor that takes account of memory traffic influence to the graphics performance. In order to achieve better energy efficiency without significant performance downgrade, we studied memory access behavior for several games from multiple perspectives. We first performed characterization of off-chip memory usage for recent mobile games and observed they demand a higher main memory throughput requirement than common mobile applications. Second, we examined the memory access rate, a parameter observed to have close relation to CPU performance in several work. For different games, they can have a distinct range of the access rate. The rates also fluctuate frame by frame while playing a game. Then, we revealed the high demand of memory throughput and the memory access rate have significant influence to the time spent on accessing main memory as well as graphics performance. Hence, a general regression model was built to predict the time spent on memory access during frame rendering and it could be used for contemporary heterogeneous mobile architectures that have a shared bus accessing main memory. We also presented how our MEMCOP governor works to predict the graphics performance by adopting the regression-based model and make a DVFS decision for CPU and GPU. Our MEMCOP work improves by 18% and 9% energy efficiency on average compared to default system governor and state-of-the-art work respectively without compromising the performance and shows the importance of considering memory behavior for mobile games. The worse results performed by running the Naive memory-aware governor indicated that it is not trivial to relate memory access to graphics performance as well as energy efficiency. We further evaluated how the adopted models perform and showed the governor can be improved by considering the variation of frame workload. As future work, we plan to explore the opportunity to further improve energy efficiency taking into account the game threads scheduling as we mentioned in Section 2.5.

Chapter 3

The Case for Exploiting Underutilized Resources in Heterogeneous Mobile Architectures

3.1 Introduction

Modern mobile platforms are being increasingly used for a variety of applications with differing computational demands, such as video, gaming, virtual/augmented reality, and computer vision applications. Consequently mobile platforms deploy Multiprocessor SoCs (MPSoCs) that integrate multiple heterogeneous CPUs together with accelerators such as GPUs, DSPs, neural processing units (NPUs) and other hardware IP blocks into a single chip (Table 3.1) to meet the performance and quality requirements of emerging applications. These accelerators are shipped with software toolchains allowing application developers to exploit domain-specific acceleration of task kernels. For instance, contemporary mobile GPUs are usually shipped with support for applications written in OpenCL and CUDA, and excel in ac-

celerating data-parallel kernels typically found in computer vision and gaming applications. Similarly, mobile DSPs come with SDK toolchains supporting application development (e.g., OpenCL for TI's Tesla DSP and C++ Qualcomm's Hexagon DSP SDK) and acceleration for signal processing applications.

Vendor	SoC	CPU	GPU	Other IPs
Qualcomm	Snapdragon	HMP	Adreno	Hexagon DSP
TI	OMAP	HMP	PowerVR	Tesla DSP
NVIDIA	Tegra	HMP	NVIDIA	-
Samsung	Exynos	HMP	Mali	Neural Processor
Apple	A series	HMP	Apple	Neural Processor

Table 3.1: Contemporary Mobile SoCs

To improve performance, a developer typically partitions an application into task kernels to be executed on compute units and accelerators (e.g., CPU, GPU, DSP) that correspondingly promise a boost in performance. For instance, a convolutional neural network (CNN) application with multiple layers can be partitioned into data-parallel tasks for each layer and mapped onto GPUs for boosting performance. Intuitively, this strict partitioning of tasks to execute them on the highest-performing compute units should result in overall better performance. However, mobile platforms often face resource contention when executing multiple applications, saturating these high-performing compute units. In such scenarios – contrary to intuition – offloading of computational pressure to other underutilized and seemingly under-performing compute units (e.g., DSPs) can actually result in overall improvements in performance and energy. Indeed, in our experimental case study, we observed an average improvement of 15-46% in performance and 18-80% in energy when executing multiple CNNs, computer vision and graphics applications on a mobile Snapdragon 835 platform by utilizing idle resources such as DSPs and considering all available resources holistically.

To estimate the underutilization of accelerators, we surveyed 175 top-ranked Android applications from popular categories such as games, video/audio players, social media and photography. Our study of these applications show that approximately 15% of them use

the GPU, while only one utilizes the DSP. These low utilization numbers motivate the opportunity to achieve better performance and energy for mobile applications by holistically exploiting underutilized resources.

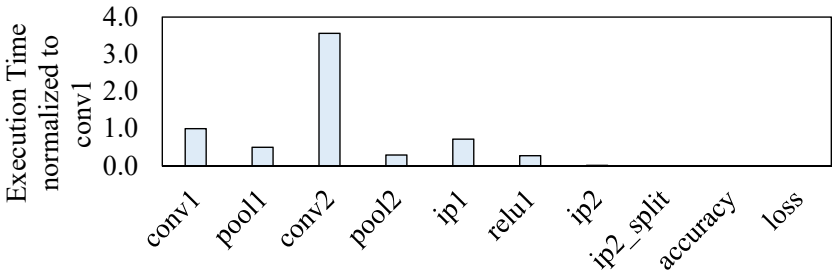
The main contribution of this chapter is a case study of executing multiple applications (e.g., CNN, computer vision and graphics) on a mobile Snapdragon 835 platform, and demonstrating the ability to exploit underutilized heterogeneous compute resources for concomitant improvements in performance and energy. In particular:

1. We demonstrate the ability to exploit underutilized mobile platform resources such as DSPs for holistic performance and energy improvement
2. For single, multiple, and mixed-workload applications, we observed between 15-45% improvements in performance, and between 18-80% improvements in energy over the conventional mapping strategy that assigns tasks to the highest-performing compute units (GPUs).

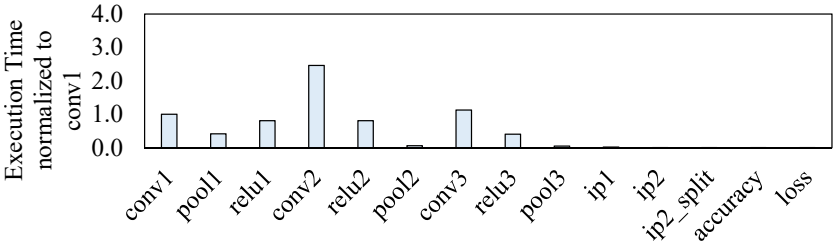
3.2 Case Study Background

Our case study deploys a popular heterogeneous mobile platform – the Qualcomm 835 – that combines a octa-core big.Little CPU architecture with an Adreno GPU and a Hexagon DSP. We consider applications typically executed on GPUs for high performance (e.g., CNNs, computer vision filters and graphics); and we aim to demonstrate opportunities for both performance and energy improvement by exploiting the underutilized DSP, as well as for opportunistic mapping of applications/tasks holistically across the CPU-GPU-DSP compute spectrum. We first present a detailed case study executing a single class of applications (e.g., CNN, canny edge detector, or a graphics application). Using the CNN as a single-application exemplar: we execute a single CNN layer, a whole CNN application split into

layer-level tasks, and multiple instances of the CNN applications to saturate the mobile platform resources. For each set of experiments, we partition and allocate the tasks across the CPU, GPU and DSP compute resources and report the performance and energy gains achieved over a traditional CPU/GPU-only mapping scheme. A similar study was performed for the canny edge detector. We then present a detailed case study of simultaneously executing multiple instances, and varying combinations of different applications/tasks (e.g., CNN, canny edge detector, and graphics) to emulate a mixed workload on a mobile platform; and repeat the above experiments to report performance and energy gains achieved by exploiting underutilized resources. For completeness we now give a short background on these applications, as well as features of the DSP that can be exploited for improving performance and energy.



(a) MNIST



(b) CIFAR10

Figure 3.1: Performance breakdown of CNN

3.2.1 Convolutional Neural Networks (CNNs)

CNNs requires a significant amount of computation for both training and inference phases, which makes them difficult to run on mobile platforms. Figure 3.1 shows the performance breakdown of inference phase for datasets MNIST [37] and CIFAR10 [38] within the Caffe [39] framework. The bottleneck is usually at convolutional layers e.g., conv1, conv2 and conv3, whose operations are usually turned into matrix multiplications. Quantized neural networks (QNN) have been developed recently, which replace input data and kernel weights by a compact data format, e.g., 8-bit fixed-point. This results in a smaller model file size for storage, less-intensive computation compared to floating-point computation, and less memory bandwidth – all while still maintaining high accuracy [40]. Hence, it is a promising replacement for standard CNNs and also more feasible for resource-constrained mobile platforms. In our experiments, we targeted the inference phase of CNN applications and executed both the original and quantized models to demonstrate the viability of using the DSP as an energy-efficient computation alternative during saturated platform execution. In our experiments, we modified the CNN's convolutional and fully-connected layers to apply quantized computation (since matrix multiplication is usually the bottleneck of the network) and left all other layers unmodified (i.e., performing floating-point arithmetic).

3.2.2 Canny Edge Detector (CED) and graphics applications

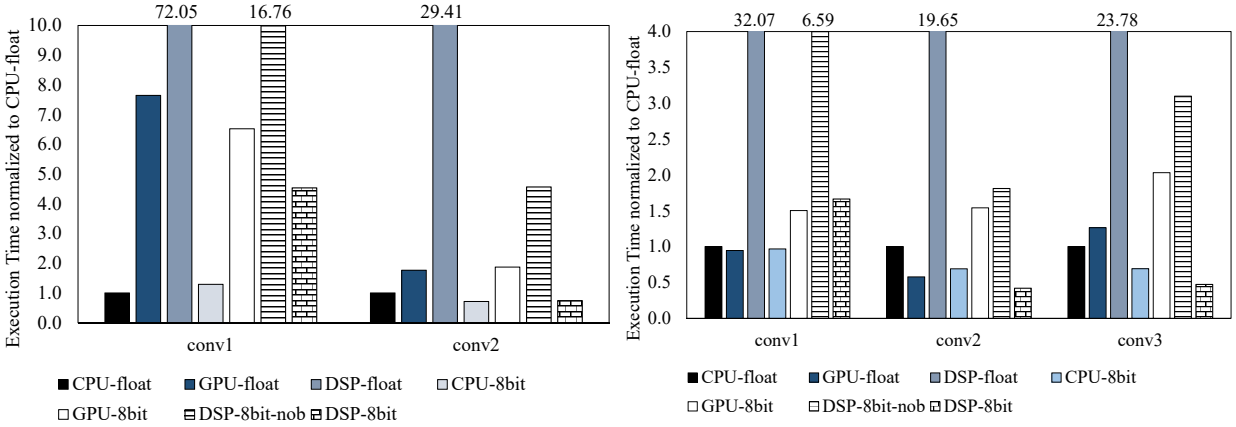
CED [41] is a well-known multi-stage edge detection algorithm. We deploy a CED implementation composed of four stages. The first two stages are the Gaussian and Sobel filters that reduce image noise as well as compute edge gradient and directions, respectively. The third stage is non-maximum suppression (nmaxsup), which finds the largest edges and suppresses nearby pixels pointing at the same direction. The fourth stage (hysteresis) removes unrelated edges. The four stages execute sequentially, each accelerated by a data parallel

computation kernel. GPU Performance Analyzer [42] is an Android application that generates intensive GPU graphics workload with user-configured graphics parameters. We execute this application for graphics workload to exercise the GPU in our experiments.

3.2.3 Qualcomm's Hexagon DSP

Qualcomm's Hexagon DSP [15] with Hexagon Vector Extension (HVX) supports very wide SIMD fixed-point operations (up to 4096 bits per cycle) and is designed to serve as an energy-efficient computing alternative to power-hungry compute units such as the CPU and GPU for some modern computer vision applications. However, HVX does not support SIMD floating-point arithmetic in order to reduce size and power consumption. Since the DSP is fixed-point optimized, it is suitable for executing applications with massive fixed-point arithmetic parallelism (e.g., quantized CNNs).

3.3 Experimental Case Study



(a) MNIST

(b) CIFAR10

Figure 3.2: Performance of Convolutional Layers

3.3.1 Experimental Setup

Experiment	Description
CPU-float, CPU-8bit	Run the original or quantized version on the CPU
GPU-float, GPU-8bit	Run the original or quantized version on the GPU
DSP-float	Run the original version on the DSP
DSP-8bit	Run the quantized version on the DSP w/ batch processing
DSP-8bit-nob	DSP-8bit w/o batch processing
Hetero	Layers or stages are statically configured to run on highest-performing compute unit
Hetero-noGPU	Like Hetero but avoid using GPU

Table 3.2: Keywords used in Experiments

Platform: We use a Snapdragon 835 development board with the Android 6 operating system (which uses the Linux 4.4.63 kernel). The board's SoC integrates custom CPUs with big-LITTLE configurations that conform to ARM's ISA. It also integrates a GPU with unified shaders, all capable of running compute and graphics workloads. The 835 board has two Hexagon DSPs: a cellular modem DSP dedicated to signal processing, and a compute DSP for audio, sensor, and general purpose processing. We target exploiting the compute DSP since it is typically idle. We use the Trepn profiler [43] to measure the power and energy consumption.

Applications: For the CNN applications, we select two Caffe CNNs: *lenet-5* and *cuda-convnet* using datasets MNIST and CIFAR10, respectively. MNIST represents a lightweight network with a few layers and low memory footprint whereas CIFAR10 has more layers and high memory footprint. We also implemented a quantized version of Caffe, which supports quantized matrix multiplication using 8-bit fixed-point for convolutional and fully-connected layers. The other layers still perform floating-point computation. The experiments include floating-point and fixed-point versions of CNN models running on CPU, GPU and DSP. For the CED application, we modified Chai CED[44] to support all heterogeneous compute resources for each stage.

Table 3.2 summarizes the different experiments by executing the above applications on various compute units (CPU, GPU, DSP, and heterogeneous – including all compute units). In addition to the original floating-point version of CNNs, we also deploy 8-bit quantized versions to exploit the DSP effectively. The row *DSP-8-bit* represents a single function call for batch processing of 100 images to amortize the communication overhead, whereas the row *DSP-8bit-nob* represents no batch processing, i.e., separate function calls for each image.

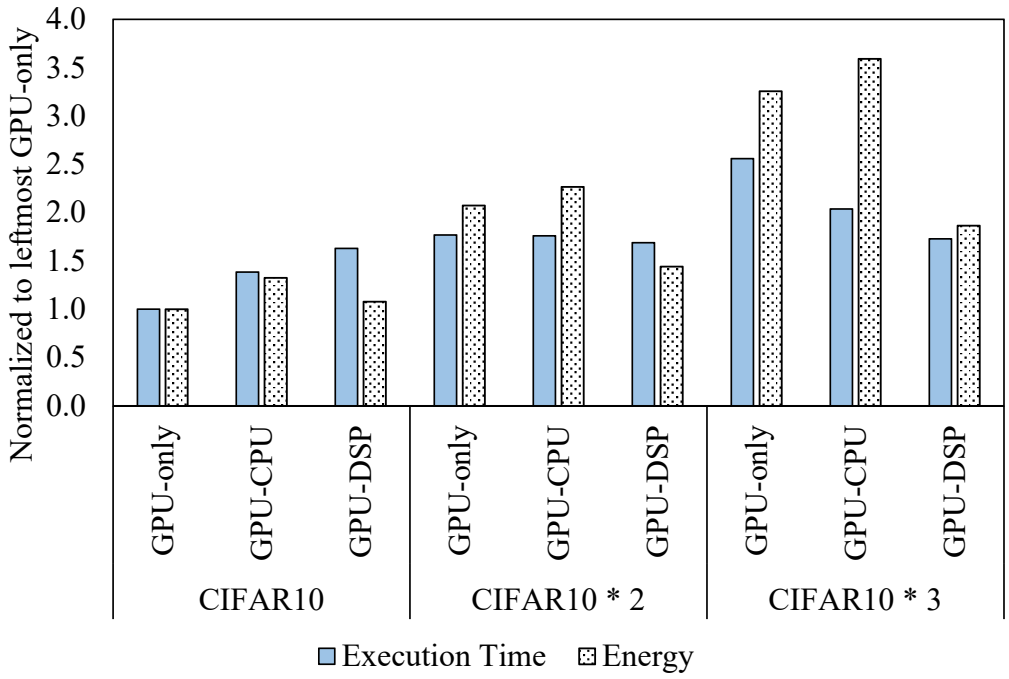


Figure 3.3: Performance of executing multiple CIFAR10 instances on different compute units

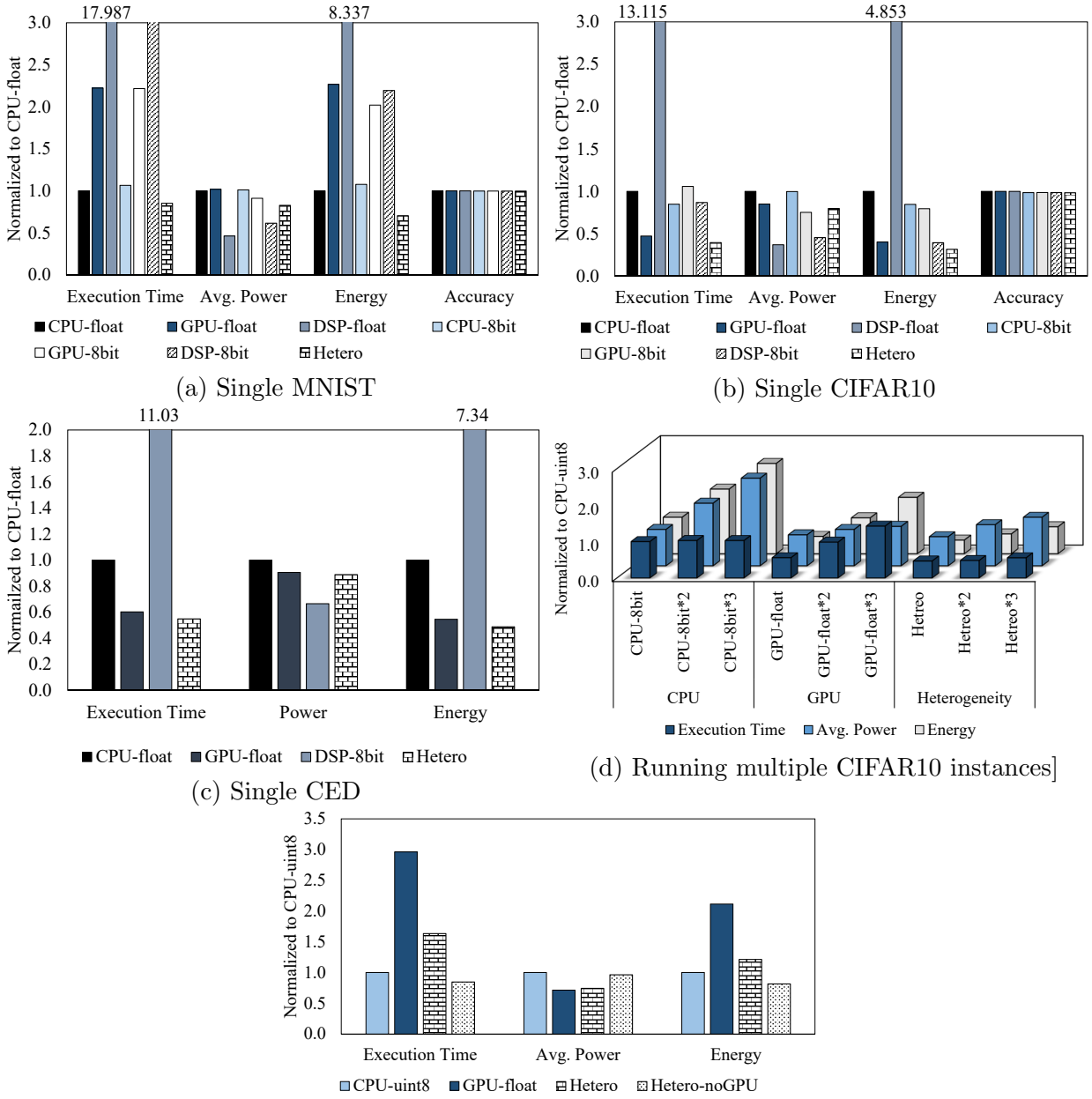
3.3.2 Opportunities for Exploiting Underutilized Resources

Figure 3.2 presents the performance of the convolutional layers of MNIST and CIFAR10. Since the Hexagon DSP is fixed-point optimized, the quantized version (*DSP-8bit*) of the conventional layers are able to outperform some of the other versions. Therefore – following intuition – the performance of a single application can be boosted by allocating the workload to the corresponding highest-performing compute unit. *However – counterintuitively – we may be able to exploit seemingly slower compute units to gain overall performance and energy*

improvements. Figure 3.3 illustrates this scenario, showing the execution time of running one to three instances of CIFAR10 in parallel. When executing only one CIFAR10 instance, the *GPU-only* version yields the best result compared to *GPU-CPU* and *GPU-DSP* versions (as expected). However, when we execute multiple instances of CIFAR10 (i.e., panels showing *CIFAR10*2* and *CIFAR10*3*), we observe that offloading to the other seemingly inferior compute units (e.g., CPU & DSP) yields overall better performance. Indeed, when executing 3 instances of CIFAR10 (*CIFAR10*3*), we see that the performance of *GPU-CPU* and *GPU-DSP* significantly outperform the *GPU-only* version, since the GPU is saturated. This simple example motivates the opportunity to exploit underutilized resources such as DSPs as outlined in Sections 3.3.3 and 3.3.4.

3.3.3 Optimization for Single Application Class

Intuitively, the performance and energy consumption of an application (e.g., CNN) can be improved by partitioning and executing on specific accelerators (e.g., GPUs). But frameworks such as Tensorflow and Caffe run the CNN model on the same GPU, saturating that compute unit while missing the opportunity to improve performance and energy consumption by exploiting other underutilized compute units (e.g., CPU and DSP). Therefore, we partition the neural network at the layer level so each layer can be executed as a task running on a different compute unit to exploit heterogeneity. Figure 3.4a, 3.4b shows the execution time, average power and energy consumption of running different versions of MNIST and CIFAR10. For MNIST, *conv2* runs on DSP and the others run on CPU. For CIFAR10, *conv2*, and *conv3* run on DSP, and the others run on GPU. Although *DSP-8bit* has better performance over convolution layers in general as shown in Figure 3.2, it performs worse due to the floating-point computation in other layers such as the Pooling and ReLu layers. For all quantized models, the accuracy drops 1.4% on average. *Hetero* represents the results of utilizing diverse compute units to gain performance and energy improvements. Indeed,



(a) Single MNIST

(b) Single CIFAR10

(c) Single CED

(d) Running multiple CIFAR10 instances]

(e) Running CIFAR10 with graphics applications

Figure 3.4: Performance, power, and energy consumption for single or multiple CNNs/CED with different task mapping

the *Hetero* results show a 15.6% performance boost and a 25.4% energy saving on average compared to *CPU-float* and *GPU-float* (which respectively perform best for MNIST and CIFAR10). We observed similar patterns for the CED experiments as shown in Figure 3.4c: the *Hetero* configuration for CED executing four stages on the DSP, GPU, CPU, CPU re-

spectively, produced the best performance and energy consumption results, demonstrating the potential to effectively exploit underutilized heterogeneous resources.

Figure 3.4d shows the results of running multiple CIFAR10 instances. The results are grouped by CPU, GPU and heterogeneous resources and the values are normalized to *CPU-8bit*. For *CPU-8bit*, the performance is scalable but the power and energy consumption increases drastically with more instances because more cores are exercised. The performance of *GPU-8bit* downgrades along with the increase of instances because they contend for the GPU. *Hetero* shows more stability than the others due to the distribution of the workload over all compute resources. We also simulate the scenario when the GPU is saturated by rendering high-quality graphics. We use the GPU Performance Analyzer benchmark to produce a high quality graphics workload. As Figure 3.4e shows, the performance of GPU-float and *Hetero* decreased significantly because the GPU is fully-saturated by the above-mentioned graphics workload. *Hetero-noGPU* is statically configured to offload the *conv2*, *conv3* and *relu* layers to DSP while the other layers run on CPU. As *Hetero-noGPU* specifically avoided using the GPU, its performance and energy consumption outperforms the others.

3.3.4 Optimization for Multiple Application Classes

When executing multiple application classes on a system, both the task partitioning and the exploitation of heterogeneous resources help for better distribution of workload, which in turn leads to better performance and energy consumption.

Figure 3.5 presents the results of running different combinations of canny edge detector (CED), CIFAR10, and the graphics application. *CPU/CPU* means CED runs the CPU and CIFAR10 also runs on the CPU. The other terms in the figure follow the same convention. Execution time is from when we execute all the applications in parallel to when the last application terminates. We make the following observations:

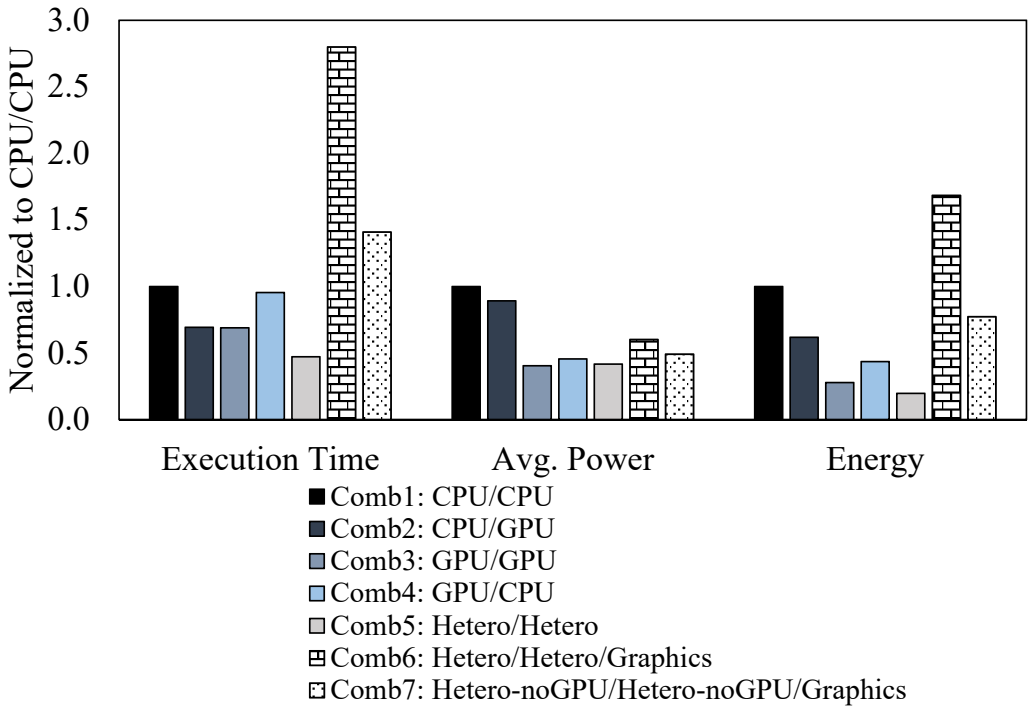


Figure 3.5: Scenarios of running multiple applications

- By exploiting all heterogeneous (including underutilized) resources efficiently, we can achieve better results: the fully heterogeneous *Comb5* outperforms GPU-only *Comb3* by 32% for performance and 29% for energy consumption.

- Underutilized resources may not be beneficial when compute units are not saturated. Consider CPU/GPU *Comb2* versus GPU-only *Comb3*: although *Comb3* executes both applications on the GPU, it is not saturated and thus still yields better results than *Comb2* that distributes utilization among heterogeneous resources.

- The Quality of service (QoS) for each application must also be considered. For instance, although the no-GPU *Comb7* has overall better performance than GPU-saturated *Comb6*, the graphics application suffers a QoS-loss of 8.9% in frame rate when switching from *Comb6* to *Comb7*, showing that we must also respect each application's QoS requirement during task mapping.

3.4 Implementation

We use Caffe as our base framework as it supports OpenCL for GPU acceleration. Note that we did consider Tensorflow as well, especially given its support for quantization. However, we did not select it since it still lacks support for OpenCL, which limits the users from accelerating it on GPUs from different vendors.

We build a quantized version of Caffe. To do this, we integrate several software components. The major effort is to make Caffe support quantized models and operations including CPU, GPU and DSP versions. We integrate Ristretto from Gysel et al. [40], which is a approximation framework built on top of Caffe with support for quantization for hardware accelerators. We use this framework to train the neural network and generate new parameters for the convolutional and fully-connected layers. In our quantized Caffe, we still store the input data and weights by 32-bit floating-point format and quantize them when necessary. We apply the quantization scheme in gemmlowp [45] to perform quantized matrix multiplication (MATMUL). It quantizes 32-bit floating-point into 8-bit integer and perform MATMUL. We also use gemmlowp library to perform quantized MATMUL for CPU. Custom kernels for MATMUL are built for GPU. We integrate nnlib [46], a library developed for Hexagon DSP to perform neural network computation.

3.5 Related Work

Heterogeneous architectures have been researched from diverse perspectives including performance, power and energy management for a wide spectrum of application domains, including CPU-GPU collaboration [47][48][49][50][51]. and CPU-DSP collaboration [52]. There is a large body of work on accelerating convolutional neural networks (CNNs) by offloading to hardware accelerators [53][54][55].

Our work differs from the above in multiple ways. First, we focus on mobile energy-constrained heterogeneous platforms and aim to holistically deploy execution of applications that can execute across multiple heterogeneous compute resources including CPU, GPU and DSP (as opposed to only one or a pair of resources). Second, we experimentally demonstrate simultaneous improvements in performance and energy through task mapping that efficiently exploits underutilized heterogeneous resources (e.g., the DSP on the mobile Snapdragon 835 platform).

3.6 Conclusion

In this chapter, we presented a case study using CNN, computer vision, and graphics applications to demonstrate the ability to exploit available yet underutilized heterogeneous resources to improve both performance and energy in mobile devices. We illustrated the potential for exploiting the DSP for such applications as this accelerator can be an efficient compute alternative that is generally underutilized. We examined different application scenarios to show the benefit of having higher heterogeneity in SoCs. For single and multiple application scenarios executing mixed workloads, we observed an average performance and energy consumption improvement of 15-46% and 18-80%, respectively, by synergistically deploying all available compute resources, especially the underutilized DSP. The performance and energy consumption turn out to be further improved when all the available compute resources are considered for the computation.

Chapter 4

SURF: Self-aware Unified Runtime Framework for Parallel Programs on Heterogeneous Mobile Architectures

4.1 Introduction

Mobile computing has benefited from a virtuous cycle of powerful computational platforms enabling new mobile applications, which in turn create the demand for ever more powerful computational platforms. In particular contemporary mobile platforms are increasingly integrating a diverse set of heterogeneous computing units [1] that can be used to accelerate newer mobile applications (e.g., augmented reality, image recognition, inferencing, 3-D gaming, etc.) that are computationally demanding. The privacy and security needs of these mobile applications (i.e., safely compute on the mobile platform, rather than suffer the vulnerability of sending to the cloud for processing) place further computational stress

[1] In the chapter we use the terms "compute unit" and "device" interchangeably

on emerging mobile platforms. Consequently, as shown in Table 3.1, contemporary mobile platforms typically include a diverse set of compute units such as multiple heterogeneous multi-processors (HMPs), and programmable accelerators such as GPUs, DSPs, NPUs, as well as other custom application-specific hardware accelerators.

However, current mobile platforms and their supporting software infrastructures are unable to fully exploit these heterogeneous compute units for two reasons: 1) existing runtime systems are typically designed for one or a few compute units, thus unable to exploit other heterogeneous compute units that are left idle, and 2) conventional wisdom dictates that certain application codes are best accelerated by specific compute units (e.g., embarassingly parallel codes by GPUs, and filtering/signal processing by DSPs). Consequently, some compute units (e.g., GPUs) can get heavily overloaded with high resource contention resulting in overall poor performance. Indeed, in our recent study [56], we made the case for exploiting underutilized resources in heterogeneous mobile architectures to gain better performance and power; and even counterintuitively using a slower/less efficient but underused compute unit to gain overall performance and power benefits when the platform is saturated. To fully exploit such situations, we postulated that there is a need for a unified runtime framework for parallel programs that can accept applications and dynamically map them to fully utilize the available heterogeneous architectures.

Towards that end, this chapter presents the software architecture and preliminary evaluation of **SURF**, our Self-aware Unified Runtime Framework for parallel programs, that exploits the range of mobile heterogeneous compute units. SURF is a unified framework built on top of existing parallel programming interfaces to provide resource management and task schedulability for heterogeneous mobile platforms. Using SURF application interfaces, application designers can accelerate application blocks by creating schedulable SURF tasks. The SURF runtime system includes a self-aware task mapping module that considers resource contention, the platform's native scheduling scheme, and hardware architecture to perform

performance-centric task mapping. We have implemented SURF in Android on a Qualcomm Snapdragon 835 development board, supporting OpenMP, OpenCL and Hexagon SDK as the programming interfaces to program CPU, GPU and DSP respectively. Our initial experimental results – using a naive, but self-aware scheduling scheme – shows that SURF achieves average performance improvements of 24% over contemporary runtime systems, when the system is saturated with mutliple applications. We believe this demonstrates the potential upside of even larger performance improvements when more sophisticated scheduling algorithms are deployed within SURF.

The rest of the chapter is organized as follows. Section 4.2 presents background on existing mobile programming frameworks and opportunities to exploit heterogeneous compute units for mobile parallel workloads. Section 4.3 presents SURF's software architecture. Section 4.4 presents early experimental results using SURF to execute sample mobile workloads. Section 4.5 discusses related work and Section 4.6 concludes the chapter.

4.2 Background

Modern mobile heterogeneous system-on-Chip (SoC) platforms are typically shipped with supporting software packages to program the integrated heterogeneous hardware accelerators. However, there is no unified programming framework. Open Computing Language (OpenCL) was designed to serve this purpose but it ends up being mostly limited to GPU only among mobile platforms. Other compute units such as DSP or FPGA need their own software supporting packages instead of relying on OpenCL. As a consequence, existing infrastructures require a static mapping of the workload to compute units at compile time. Static mapping of tasks to specific compute units (e.g., data parallel tasks to the GPU) can result in severe resource contention for one unit (e.g., the GPU) while underutilizing other units (e.g., DSP). Besides, there is no information sharing between individual device

runtimes, which makes it difficult to make intelligent task-mapping decisions even if the schedulability is provided. Hence, existing software infrastructures are unable to exploit the full heterogeneity of compute units. In our previous case study [56], we showed how underutilized heterogeneous resources can be exploited to boost performance and gain power saving when the platform is saturated with workloads – an increasingly common scenario for mobile platforms where users are multi-tasking between mobile games, image/photo manipulation, video streaming, AR, etc. Our study highlighted the need for a new runtime that can dynamically manage and map applications to heterogeneous resources at runtime. To address these challenges, we have built SURF, a unified framework that sits on top of existing parallel programming interfaces to provide resource management and task schedulability for mobile heterogeneous platforms. Using SURF application interfaces, application designers can accelerate application blocks by creating schedulable SURF tasks. Next we analyze the performance of several popular mobile data parallel workloads on heterogeneous compute units to illustrate the potential for SURF to map these computations across these units.

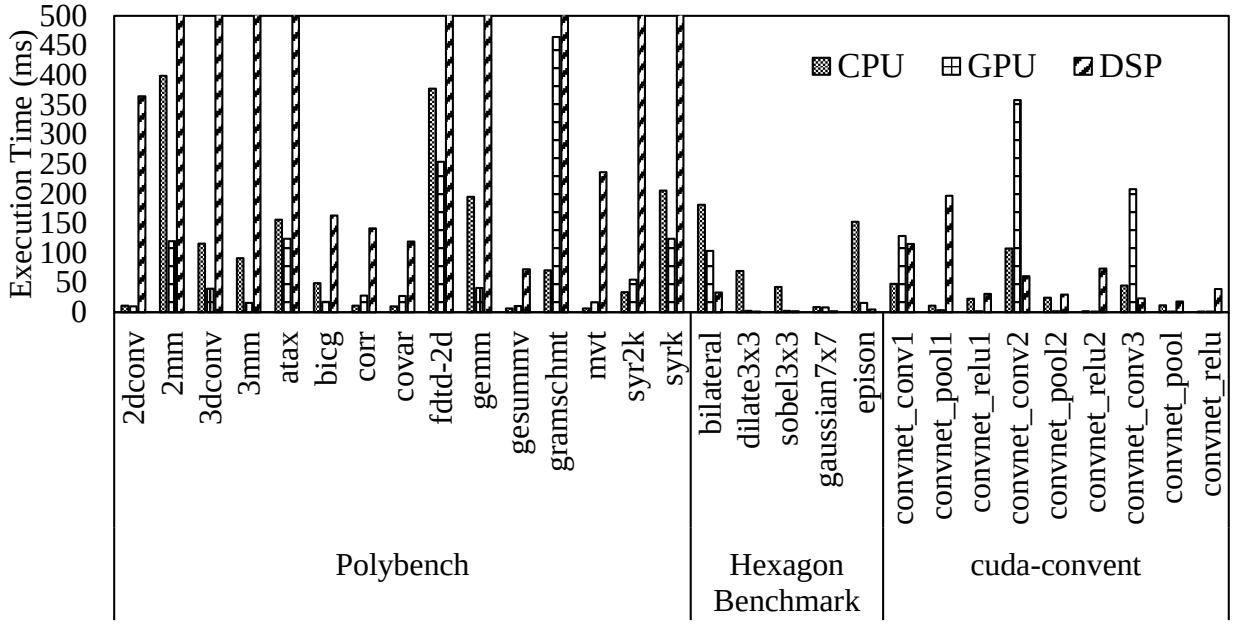


Figure 4.1: Execution time of benchmarks on different compute units

Data-Parallel Workload Characterization Data-parallel computations are common in several mobile application domains such as image recognition (using CNNs) and image/video processing/manipulation where the same function is applied to a huge amount of data. Due to the simplicity of this programming pattern, they can be easily offloaded to hardware accelerators such as GPUs without substantial programming effort. In order to highlight the opportunity for gaining performance improvement through task mapping/schedulability across heterogeneous compute units, we measured the execution time of two benchmark suites (Polybench benchmark suite [57] and Hexagon SDK benchmark suite [58]), as well as for the critical layers in a CNN (cuda-convnet) that contain several common data-parallel kernels across different domains. In addition to their original implementations, we added OpenMP/CPU, OpenCL/GPU or C/DSP implementations to execute them on different compute units (CPU, GPU, DSP).

Figure 4.1 shows the measurement results of running each benchmark on the CPU, GPU and DSP respectively. As expected, we typically see one "dominant" version for best performance on a specific compute unit, e.g., syrk and convnet_pool1 have the lowest execution time on GPU, whereas bilateral and convnet_conv2 runs best on the DSP. However, note that the non-dominant (slower) versions (e.g., syrk and convnet_pool1 on CPU or DSP; and bilateral and convnet_conv2 on CPU or GPU) – while seemingly inferior in performance – can be opportunistically exploited by our SURF runtime to improve overall system performance, especially as the mobile platform suffers from high contention when popular apps (e.g., image recognition, photo manipulation/filtering) compete for a specific compute unit (e.g., the GPU for data parallel computations).

4.3 SURF: Self-aware Unified Runtime Framework

SURF is a unified runtime framework built on top of existing programming interfaces and device runtime to provide adaptive, opportunistic resource management and task schedulability that exploits underutilized compute resources. Figure 4.2 shows the architectural overview of SURF. In a nutshell, mobile applications create SURF tasks through SURF APIs. When a SURF task is submitted, a self-aware task mapping algorithm is invoked referencing runtime information of compute units provided by SURF service. After the task mapping decision is made, the corresponding parallel runtime stub executes that task.

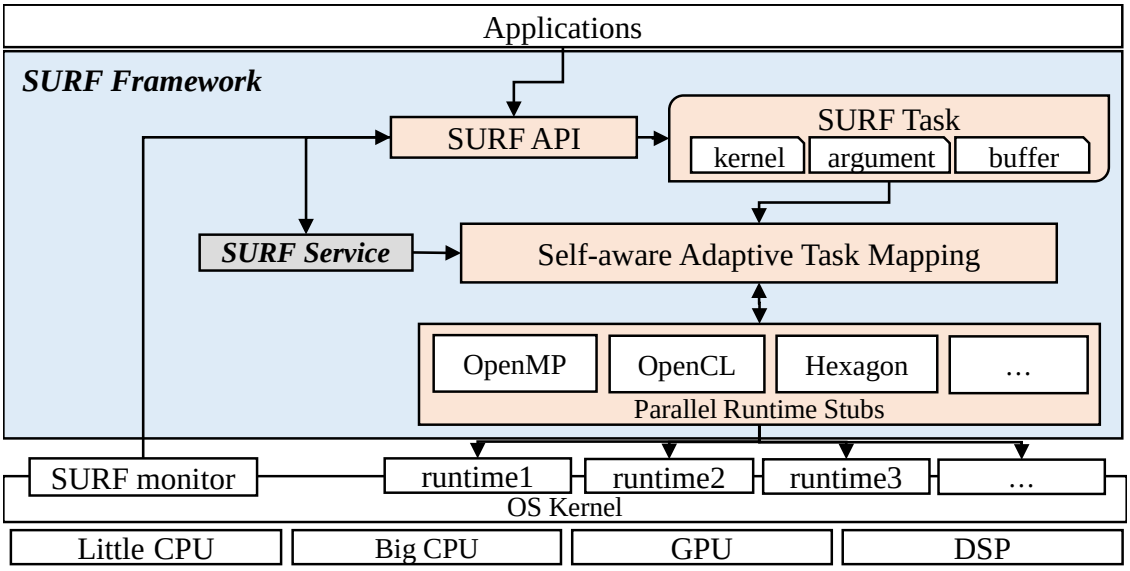


Figure 4.2: SURF Architecture

4.3.1 Application and Task Model

Figure 4.3 shows the hierarchy of SURF's application model. At the highest level, the mobile platform admits new applications at any time. A newly entering application (e.g., CNN in Figure 4.3) can create and submit tasks to SURF dynamically. A task (e.g., conv1, pool and relu1 in Figure 4.3's CNN application) represents a computational chunk (parallel algorithm or application block) that could be a candidate for acceleration. A kernel residing in a

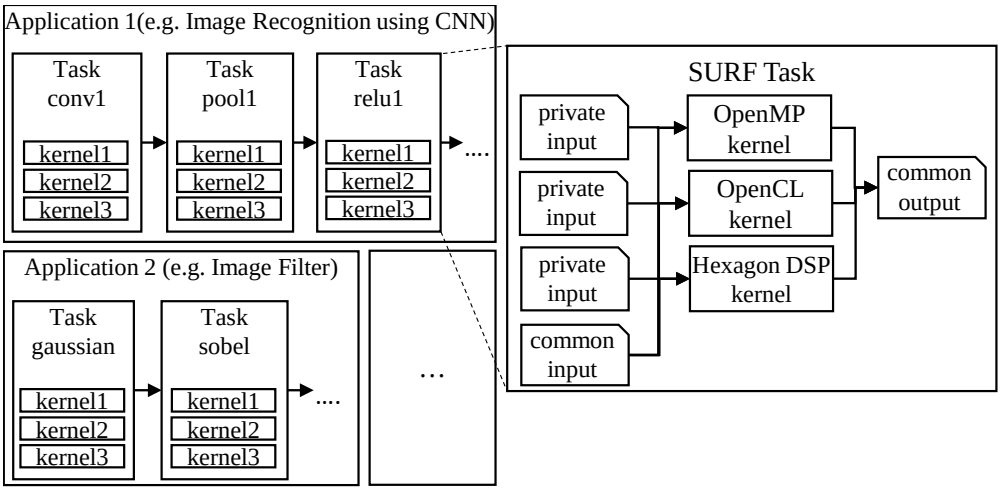


Figure 4.3: Application and Task Model

task represents the programming-interface-specific implementation artifact to program one compute unit (e.g., OpenMP, OpenCL and Hexagon DSP kernels as shown on the right side of Figure 4.3). SURF opportunistically maps each task (encapsulating multiple kernels) for scheduling execution on a specific compute unit. All kernels in a task share a set of common inputs and outputs, as well as a set of private inputs that are specific to the corresponding programming interface.

The code block in Figure 4.4 demonstrates an example of how to use the application interfaces to create and execute a 2-dimensional convolution task with three kernels including an OpenMP, a OpenCL and a Hexagon DSP kernel. Lines 1-2 create the input and output SURF buffer; Line 4 creates a task; Lines 5-8 add common arguments for all kernels; Lines 9-11 create three kernels to run on CPU, GPU and DSP with user-provided OpenMP binary, OpenCL source code, Hexagon DSP binary respectively, and associate the kernels with the task; and Lines 11-12 execute and destroy the task.

```
1  surf_buffer_t in  = surf_buffer_create(size_in);
2  surf_buffer_t out = surf_buffer_create(size_out);
3  /* fill in input buffer */
4  surf_task_t task = surf_task_create(3);
5  surf_task_add_args(task, 0, in, size_in, SURF_MEM_READ | SURF_MEM_BUFFER);
6  surf_task_add_args(task, 1, out, size_out, SURF_MEM_WRITE | SURF_MEM_BUFFER);
7  surf_task_add_args(task, 2, &ni, sizeof(int), 0);
8  surf_task_add_args(task, 3, &nj, sizeof(int), 0);
9  surf_task_create_kernel (task, "conv2D_cpu", SURF_DEV_CPU,
       SURF_KERNEL_OPENMP | SURF_KERNEL_USE_BINARY, "res/libpb.so", 0);
10 surf_task_create_kernel (task, "conv2D_gpu", SURF_DEV_GPU,
       SURF_KERNEL_OPENCL | SURF_KERNEL_USE_SOURCE, "res/2dconv.cl", 0);
11 surf_task_create_kernel (task, "conv2D_dsp", SURF_DEV_DSP,
       SURF_KERNEL_HEXAGON | SURF_KERNEL_USE_BINARY, "res/libconv.so", 0);
12 surf_task_enqueue(task);
13 surf_task_destroy(task);
```

Figure 4.4: Sample code of SURF application interfaces including SURF buffer, task and kernel creation as well as SURF task execution and termination.

4.3.2 Memory Management and Synchronization

SURF assumes compute units are sharing the system memory which is also the dominant architecture in mobile SoCs. Hence, the expensive data movement between device memory can be ignored if the memory is mapped to all the devices correctly. The SURF buffer object is a memory region mapped to all the device address space through device-specific programming interfaces e.g., OpenCL Qualcomm extension and Hexagon SDK APIs for Qualcomm SoCs. Memory synchronization is still necessary when the buffer is used among different devices to ensure the running device can see the most recent update of data. SURF automatically synchronizes memory objects when the memory object is going to be used by a different device; this memory overhead is included in SURF's task mapping decision.

4.3.3 Self-aware adaptive task mapping

SURF employs a self-aware adaptive task mapping strategy. SURF exhibits self-awareness [59] by creating a model of the underlying heterogeneous resources, assessing current system state

via the SURF monitor, and using predictive models to guide mapping decisions. This enables SURF to act in a self-aware manner, combining both *reactive* (e.g., as new applications arrive or when active applications exit), as well as *proactive* (e.g., through the use of predictive models to enable evaluation of opportunistic mapping to underutilized compute units) strategies to enable efficient, adaptive runtime mapping.

SURF's current implementation deploys a variant of the heterogeneous earliest finish time (HEFT) [60] task mapping algorithm, enhanced to incorporate the cost of runtime resource contention. We consider two types of contention:

intra-compute-unit the contention happens when multiple tasks are submitted to a compute unit. The cost of the contention depends on the device runtime and the hardware architecture. For compute unit accelerators such as GPU and DSP, the task execution is usually exclusive due to costly context switch overheads. A FIFO task queue is implemented for each compute unit, so we include the wait time in the queue when calculating the finish time for a task. We also consider device concurrency (i.e., how many tasks can run concurrently on a device) in the analysis. Contemporary mobile GPUs can only accommodate one task execution at a time. Other devices such as DSPs may have more than one concurrent task execution (e.g. Qualcomm Hexagon DSP supports up to 2 when setting to 128-byte vector context mode [58]). And of course for the CPU cluster we can have multiple, concurrent tasks executing across the big.LITTLE cores, that typically employs an existing sophisticated scheduler such as the Linux Completely Fair Scheduler (CFS) [61].

inter-compute-unit Typically memory contention is the major bottleneck when there are concurrent memory-intensive task executions in different compute units, resulting in the execution makespan of a task increasing significantly.

SURF proposes a heuristic-based scheme to estimate the finish time for a task running on

different compute units considering both intra- and inter-compute-unit contention. First, to determine which compute unit has the fastest execution time, a new task starts within a profile phase to measure the execution time for all kernels in the task. Map phase comes after the profile phase is finished where it begins to find the earliest finish time based on the runtime information. Equation 4.1 shows how we estimate the finish time. T^{cu}_{task} is the finish time when executing task t on compute unit cu. T_{inter} is the execution time considering inter-compute-unit contention. The influence of memory contention to execution time is difficult to estimate at runtime because the micro-architecture metrics for hardware accelerators are usually not feasible; hence we use a history-based method to model that effect. A history buffer is introduced to track execution time of the latest n runs. T_{inter} is the average of the history buffer. T_{intra} is the execution time considering intra-compute-unit contention. For GPU/DSP, T_{intra} is the sum of execution time of earlier submitted tasks. For CPU, T_{intra} is complicated to estimate if left unbounded. So we estimate the worst execution time based on OpenMP programming model and assume the active CPU threads have the same priority under CFS policy (each thread is allocated with the same time slice). SURF configures an OpenMP kernel to execute on a CPU cluster with a thread on each core. Hence, we approximate the worst execution time by Equation 4.2. TPC is the number of concurrent OpenMP tasks in the CPU cluster. T_o represent the overhead of deploying the task to the compute units and the memory synchronization if it is necessary (e.g., memory buffer is written by GPU and CPU is going to use the results). SURF finds the kernel with the minimum T^{cu}_{task} and submits it to the SURF device queue for execution.

$$T^{cu}_t = T^{cu}_{inter} + T^{cu}_{intra} + T_o, cu \in \{CPU, GPU, DSP\} \tag{4.1}$$

$$T_{intra}^{cpu} = TPC * T_{inter}^{cpu} \tag{4.2}$$

4.3.4 Parallel Runtime Stub

Parallel runtime stub is an abstract layer on top of the existing programming interfaces. This layer utilizes their interfaces to communication with the corresponding runtime. The corresponding stub provides the following features: a) Initialization of programming resources for different programming interfaces accordingly; b) Memory management and synchronization: while the shared system memory model between heterogeneous compute units is dominant in mobile SoCs, and saves expensive data movement, it still needs to perform memory synchronization between cache and system memory before another compute unit accesses the memory; and c) Computation kernel execution. SURF currently supports three programming interfaces: OpenMP, OpenCL and Hexagon SDK to program CPU, GPU and DSP respectively.

4.3.5 SURF Service and Monitor

The SURF service is a background process that synchronizes the system information with application processes. The SURF Monitor collects system status and profile results. For example, we collect execution time of OpenMP threads from the entity *sum_exec_runtime* through sysfs so to estimate how long an OpenMP kernel runs.

4.4 Experimental Results

4.4.1 Experimental Setup

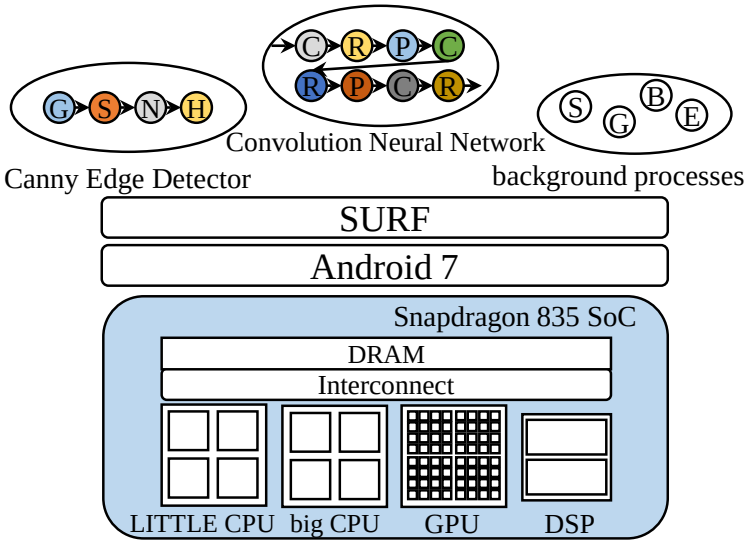


Figure 4.5: Experimental Setup

Table 4.1: Details of applications and benchmarks used in our experimental sets

Name	Source	Category	#tasks	Dominant Device
CUDA-Convnet	Caffe	Image Recognition	9	mixed
Canny Edge Detector	Synthetic	Image Filter	4	mixed
syrk	Polybench	Linear Algebra	1	GPU
gemm	Polybench	Linear Algebra	1	GPU
bilateral	Hexagon SDK	Image Filter	1	DSP
epsilon	Hexagon SDK	Image Filter	1	DSP

Figure 4.5 shows our experimental setup. We have implemented the SURF framework using C/C++ in Android 7 running on Qualcomm Snapdragon 835 development board, which has two CPU clusters (big.LITTLE configuration), and integrated GPU and DSP. SURF considers the little CPU cluster, big CPU cluster, GPU and DSP as four compute units when making task mapping decisions where GPU and DSP are exclusive for 1 and 2 tasks respectively. SURF kernels can be created by the programming interfaces of OpenMP, OpenCL

Table 4.1: Details of applications and benchmarks used in our experimental sets (continuation)

Name	#Iteration	Workload		
		Heavy(H)	Medium(M)	Light(L)
CUDA-Convnet	150	batch=100, 32x32	n/a	batch=10
Canny Edge Detector	150	batch=100, 640x354	n/a	batch=1
syrk	200	512x512	384x384	256x256
gemm	200	768x768	512x512	256x256
bilateral	200	3840x2160	1920x1080	1280x960
epsilon	200	7680x4320	3840x2160	1920x1080

and Hexagon SDK to program CPU, GPU and DSP respectively. We deploy the Caffe convolutional neural network framework [39], Canny Edge Detector (CED), Polybench benchmark suite and Hexagon SDK benchmarks to run on SURF. We also use the Snapdragon Profiler [62] to measure the utilization for each compute unit. The big.LITTLE processor governors are set to performance mode so as to not interfere with our performance-centric task mapping.

In our experimental sets, we run two applications: image recognition (cuda-convnet within Caffe and with Cifar10 dataset) and image filter (CED) representing foreground processes that have 9 and 4 SURF tasks respectively. We also run two GPU-dominant benchmarks (*syrk* and *gemm* from Polybench) and two DSP-dominant benchmarks (*bilateral* and *epsilon*) representing background processes and each of the benchmarks runs one SURF task. We characterize application workloads as heavy and light workloads by changing batch processing size (how many images are processed each iteration) and benchmark workloads as heavy, medium and light workload by changing their input size. Light workload is characterized as real-time workload which can be done within 20ms. Medium and heavy workload are the ones can be done within 20-100ms and above 100ms respectively. Table 4.1 summarizes the configurations of applications and benchmarks used in our experimental sets.

4.4.2 Experimental Results

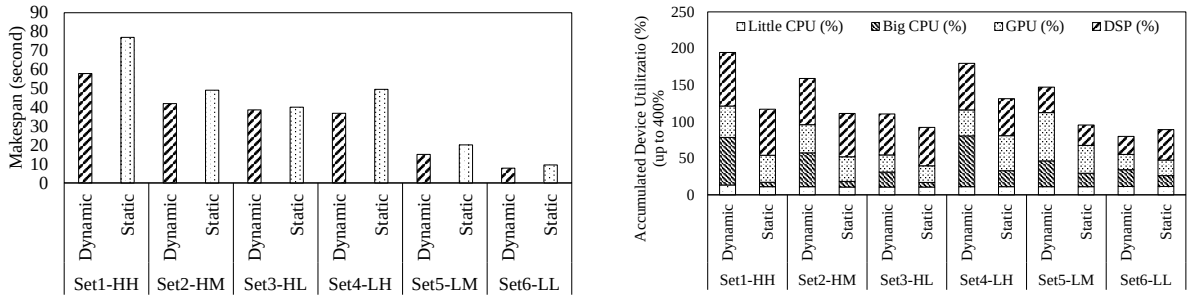
As Table 4.2 shows, we run six test sets composed of combinations of heavy/light applications and heavy/medium/light benchmarks. Figure 4.6a shows the execution makespan

Table 4.2: Speedup for different test sets

	Foreground-Background Workload	Speedup	Makespan Difference (s)
Set1	H-H	1.33	19.07
Set2	H-M	1.17	7.15
Set3	H-L	1.04	1.43
Set4	L-H	1.34	12.60
Set5	L-M	1.34	5.07
Set6	L-L	1.22	1.72

of running our six test sets with static best-performing task mapping and SURF dynamic task mapping. The static best-performing mapping configures each task to run on their best-performing compute unit according to the profiling results without SURF. SURF's dynamic task mapping outperforms static mapping by 24% on average. Table 4.2 also shows that the speedup increases with the level of the background benchmark workload because for heavy background benchmarks, a single run of them will occupy the compute resources for long time in GPU and DSP, which creates opportunities to map alternative kernels to exploit other underutilized compute units. The light applications have better speedup than heavy applications because the light application setup experiences more contention with background processes during the entire makespan and it is easy to find alternative kernels because the kernels in one task tend to have similar performance in light workload configuration. Figure 4.6b shows the sum of all device utilization of the makespan including little/big CPUs, GPU and DSP utilization when running each test set (max 400% across the 4 classes of units). GPU and DSP utilization are similar across all in general, since the GPU and DSP are heavily exercised. Here the Big CPU is better utilized by our dynamic scheme, and is the major contributor to the speedup.

While these preliminary experimental results demonstrate SURF's efficacy in exploiting underutilized compute units for improving performance, we expect to see similar reductions in energy consumption as indicated in our earlier study [56]; these performance and energy experiments are currently ongoing.



(a) Makespan of static and SURF dynamic task mapping

(b) Device utilization of makespan for each test set

Figure 4.6: SURF performance when GPU and DSP are both saturated

4.5 Related Work

Heterogeneous resource management has been widely studied, with a large body of existing work on task scheduling/mapping algorithms [60] [63] [64] [65]. For instance, Topcuoglu et al. [60] proposes the heterogeneous earliest finish time (HEFT) algorithm that schedules tasks in a directed acyclic graph (DAG) onto a device to minimize execution time. Choi et al. [64] estimates the remaining execution time for tasks on CPU and GPU by using a history buffer and selects the most suitable device. The StarPU [66] framework targets high performance computing and enables dynamic scheduling between CPU and GPU based on static knowledge of the tasks. Zhou et al. [67] perform task mapping onto heterogeneous platforms for fast completion time. Some recent efforts also address domain-specific platforms: Wen et al. [68] and Bolchini et al. [69] propose dynamic task mapping schemes specific for OpenCL; Georgiev et al. [70] proposes a memetic algorithm based task scheduler for mobile sensor workload; and Aldegheri et al. [71] presents a framework allowing multiple

programming languages and exploit their different level of parallelism for computer vision applications which achieves better performance and energy consumption.

SURF distinguishes from these works in two directions. First, the SURF framework is composed of a runtime system for task mapping and APIs for mobile systems. SURF is built on top of existing programming interfaces and dynamically profiles task execution and perform task mapping without user-provided static knowledge. Second, SURF is *self-aware*: aware of the heterogenous hardware architecture, existing scheduling scheme and the runtime system status. It takes care of resource contention of single compute units while other works make assumptions that all the compute unit are exclusive to a single task (e.g., CPU should not be exclusive). The device concurrency of hardware accelerators is also ignored in these previous works.

4.6 Conclusion

In this chapter, we presented the architecture of SURF, a self-aware unified runtime framework built on top of existing programming interfaces including OpenMP, OpenCL and Hexagon DSP SDK for mapping tasks onto CPU, GPU, and DSP respectively in mobile SoCs. We illustrated how to use SURF's application interfaces to create and execute a SURF task. SURF performs task mapping while being aware of existing scheduling schemes, intra- and inter-compute-unit contention and heterogeneous hardware architectures to select the compute unit with the earliest finish time for the given tasks without user-provided static information about the tasks. Our early experimental results show an average of 24% speedup by running mixed mobile workloads including two applications, image recognition by using convolution neural networks and an image filter with couple of background processes sharing workload on the compute units. Our ongoing work is incorporating more sophisticated mapping and prediction algorithms, and analyzing the performance as well as energy benefits of

deploying SURF on emerging heterogeneous mobile platforms.

Chapter 5

Conclusions and Future Directions

5.1 Summary and Conclusions

The prosperous mobile application market and new emerging applications have driven mobile SoCs to integrate more powerful and new hardware accelerators. The richness of on-chip compute resources opens the opportunity of providing more energy-efficiency and better performance. On the other hand, the management of these compute resources appear challenging. We identified one major problem that the current mobile systems are lack of a collaborative manner to perform power and resource management. The compute resource typically has its own runtime system which does not interact with other runtime systems. Resource-intensive mobile applications such as games, machine learning applications requires multiple compute resources to achieve acceptable performance and energy efficiency. The independent management can often lead to (1) energy inefficiency due to unawareness of application performance goals and (2) resource contention and under-utilization due to the lack of capability of task scheduling between different compute resources. This thesis describes three scenarios of collaborative runtime to address issues outlined previously: (1) MEMCOP:

a memory-aware cooperative CPU-GPU governor which perform DVFS to improve energy efficiency with satisfactory performance for mobile games. (2) A case study of executing a mix of workloads including CNN, computer vision and graphics rendering applications and gain better performance and energy-efficiency with underutilized compute resources and (3) SURF: a self-adaptive unified runtime framework for parallel applications. SURF provides programming interfaces for application developers to submit application blocks as tasks that can be accelerated by hardware accelerators. The tasks are scheduled by SURF following an adaptive heterogeneous-earliest-finish-time heuristic to boost performance specifically when the system is heavily exercised.

5.2 Future Directions

There are still several open work and challenges to enable seamless collaborative management that can be interesting directions or future work:

- Integrate MEMCOP and SURF: SURF currently has performance-centric policies which focus on compute resource utilization and application performance boost. The policy can drain the battery very fast due to unawareness of power consumption and energy efficiency. Hence, sophisticated power-aware policies are necessary to extend the battery life. For instance, the idea of MEMCOP can be integrated into SURF with the knowledge of application performance goals easily.

- Machine learning: the heuristics proposed in MEMCOP and SURF can be possibly replaced by machine learning. For example, the task scheduling problem in SURF might be formalized as a classification problem to capture the subtle difference of execution models in different compute resources without knowing what they are and reduce the complexity of building prediction models.

- Task Partitioning: data parallel applications can be easily benefit from task partitioning techniques to execute a task partially in different compute resources, specifically loop-based tasks with data-independent loops. A SURF task can be partitioned and naturally execute sub-tasks on different compute resources to have better performance or energy efficiency.

- Extensive multi-versioning and policy supervisor: the multi-versioning of SURF tasks currently address execution on heterogeneous compute resources and hence focuses on performance. The task versions can have more variety such as low accuracy fast task or high accuracy slow task. Policy supervisors can be developed to dynamically select tasks with different attributes according to the system requirement.

Bibliography

[1] Anandtech, "Qualcomm's new snapdragon s4: Msm8960 & krait architecture explored," 2011. [Online]. Available: https://www.anandtech.com/show/4940/qualcomm-new-snapdragon-s4-msm8960-krait-architecture

[2] C. Hsieh, J.-G. Park, N. Dutt, and S.-S. Lim, "Memcop: memory-aware co-operative power management governor for mobile games," *Design Automation for Embedded Systems*, vol. 22, no. 1-2, pp. 95–116, 2018.

[3] C. Hsieh, A. A. Sani, and N. Dutt, "The case for exploiting underutilized resources in heterogeneous mobile architectures," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019, pp. 1265–1268.

[4] C. Hsieh, A. A. Sani, and N. Dutt, "Surf: Self-aware unified runtime framework for parallel programs on heterogeneous mobile architectures," in *2019 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2019.

[5] Apple, "iphone." [Online]. Available: https://www.apple.com/ios/app-store/

[6] Google, "Android." [Online]. Available: https://www.android.com/

[7] Statista, "Number of available applications in the google play store from december 2009 to march 2018." [Online]. Available: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/

[8] ——, "Popular apple app store categories in may 2019." [Online]. Available: https://www.statista.com/statistics/270291/popular-categories-in-the-app-store/

[9] Techcrunch, "Digital media consumption report," 2015. [Online]. Available: https://techcrunch.com/

[10] Google, "Google play store." [Online]. Available: https://play.google.com/store

[11] Letustweak, "Snapdragon 820 features and performance," 2016. [Online]. Available: http://www.letustweak.com/tweaks/qualcomm-snapdragon-820-features-performa

[12] Qualcomm, "Tensorflow machine learning now optimized for the snapdragon 835 and hexagon 682 dsp," 2017. [Online]. Available: https://www.qualcomm.com/news/onq/2017/01/09/tensorflow-machine-learning-now-optimized-snapdragon-835-and-hexagon-682-dsp

[13] Arm, "Mali gpu." [Online]. Available: https://developer.arm.com/ip-products/graphics-and-multimedia/mali-gpus

[14] Google, "Edge tpu," 2018. [Online]. Available: https://cloud.google.com/edge-tpu/

[15] L. Codrescu, W. Anderson, S. Venkumanhanti, M. Zeng, E. Plondke, C. Koob, A. Ingle, C. Tabony, and R. Maule, "Hexagon dsp: An architecture optimized for mobile multimedia and communications," *IEEE Micro*, vol. 34, no. 2, pp. 34–43, Mar 2014.

[16] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.

[17] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, "Integrated cpu-gpu power management for 3d mobile games," in *DAC '14 Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, Jun. 2014, pp. 1–6.

[18] W.-Y. Liang, Y.-L. Chen, and M.-F. Chang, "A memory-aware energy saving algorithm with performance consideration for battery-enabled embedded systems," in *Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on*, June 2011, pp. 547–551.

[19] S. J. Cho, S. H. Yun, and J. W. Jeon, "A powersaving dvfs algorithm based on operational intensity for embedded systems," *IEICE Electronics Express*, vol. advpub, 2015.

[20] Y. Bai, "White paper memory characterization to analyze and predict multimedia performance and power in an application processor," 2011.

[21] F. Ercan, N. A. Gazala, and H. David, "An integrated approach to system-level cpu and memory energy efficiency on computing systems," in *Energy Aware Computing, 2012 International Conference on*, Dec 2012, pp. 1–6.

[22] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: ACM, 2012, pp. 850–855. [Online]. Available: http://doi.acm.org/10.1145/2228360.2228513

[23] Y. Gu, S. Chakraborty, and W. T. Ooi, "Games are up for dvfs," in *DAC '06 Proceedings of the 43rd annual Design Automation Conference*, Jul. 2006, pp. 598–603.

[24] B. Dietrich and S. Chakraborty, "Lightweight graphics instrumentation for game state-specific power management in android," *Multimedia Syst.*, vol. 20, no. 5, pp. 563–578, Oct. 2014. [Online]. Available: http://dx.doi.org/10.1007/s00530-014-0377-x

[25] F. Ercan, N. A. Gazala, and H. David, "An integrated approach to system-level cpu and memory energy efficiency on computing systems," in *Energy Aware Computing, 2012 International Conference on*, Dec 2012, pp. 1–6.

[26] D. You and K. Chung, "Quality of service-aware dynamic voltage and frequency scaling for embedded gpus," *Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2014.

[27] D. Kim, N. Jung, and H. Cha, "Content-centric display energy management for mobile devices," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 41:1–41:6. [Online]. Available: http://doi.acm.org/10.1145/2593069.2593113

[28] A. Pathania, A. E. Irimiea, A. Prakash, and T. Mitra, "Power-performance modelling of mobile gaming workloads on heterogeneous mpsocs," in *Proceedings of the 52Nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: ACM, 2015, pp. 201:1–201:6. [Online]. Available: http://doi.acm.org/10.1145/2744769.2744894

[29] Google, "Butter project," Jun. 2014. [Online]. Available: https://developers.google.com/events/io/2012/sessions/gooio2012/109/

[30] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim, "Quality-aware mobile graphics workload characterization for energy-efficient dvfs design," in *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on*, Oct 2014, pp. 70–79.

[31] D. Kadjo, U. Ogras, R. Ayoub, M. Kishinevsky, and P. Gratz, "Towards platform level power management in mobile systems," in *System-on-Chip Conference (SOCC), 2014 27th IEEE International*, Sept 2014, pp. 146–151.

[32] R. Shuvalov, "Gpu performance analyzer," May 2013. [Online]. Available: https://play.google.com/store/apps/details?id=com.romanshuvalov.gpupa

[33] Hardkernel, "Odroid-xu3," 2014. [Online]. Available: http://www.hardkernel.com/main/products/prdt_info.php

[34] L. McVoy and C. Staelin, "Lmbench: Portable tools for performance analysis," in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '96. Berkeley, CA, USA: USENIX Association, 1996, pp. 23–23. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268299.1268322

[35] B. Dietrich and S. Chakraborty, "Forget the battery, let's play games!" in *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on*, Oct 2014, pp. 1–8.

[36] B. Dietrich, D. Goswami, S. Chakraborty, A. Guha, and M. Gries, "Time series characterization of gaming workload for runtime power management," *Computers, IEEE Transactions on*, vol. 64, no. 1, pp. 260–273, Jan 2015.

[37] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, vol. 2, 2010.

[38] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[39] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22Nd ACM International Conference on Multimedia*, ser. MM '14. New York, NY, USA: ACM, 2014, pp. 675–678. [Online]. Available: http://doi.acm.org/10.1145/2647868.2654889

[40] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *CoRR*, vol. abs/1604.03168, 2016. [Online]. Available: http://arxiv.org/abs/1604.03168

[41] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov 1986.

[42] R. Shuvalov. (2013) Gpu performance analyzer. [Online]. Available: https://play.google.com/store/apps/details?id=com.romanshuvalov.gpupa

[43] Qualcomm, "Trepn power profiler." [Online]. Available: https://developer.qualcomm.com/software/trepn-power-profiler

[44] J. Gmez-Luna, I. E. Hajj, L. Chang, V. Garca-Floreszx, S. G. de Gonzalo, T. B. Jablin, A. J. Pea, and W. Hwu, "Chai: Collaborative heterogeneous applications for integrated-architectures," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2017, pp. 43–54.

[45] G. Inc. gemmlowp: a small self-contained low-precision gemm library. [Online]. Available: https://github.com/google/gemmlowp

[46] Qualcomm. (2016) nnlib. [Online]. Available: https://source.codeaurora.org/quic/hexagon_nn/nnlib

[47] J. Lee, M. Samadi, and S. Mahlke, "Orchestrating multiple data-parallel kernels on multiple devices," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, Oct 2015, pp. 355–366.

[48] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 273:273–273:283. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544163

[49] A. Prakash, S. Wang, A. E. Irimiea, and T. Mitra, "Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms," in *2015 33rd IEEE International Conference on Computer Design (ICCD)*, Oct 2015, pp. 208–215.

[50] J.-G. Park, C.-Y. Hsieh, N. Dutt, and S.-S. Lim, "Synergistic cpu-gpu frequency capping for energy-efficient mobile games," *ACM Trans. Embed. Comput. Syst.*, vol. 17, no. 2, pp. 45:1–45:24, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3145337

[51] C.-Y. Hsieh, J.-G. Park, N. Dutt, and S.-S. Lim, "Memcop: memory-aware co-operative power management governor for mobile games," *Design Automation for Embedded Systems*, Mar 2018. [Online]. Available: https://doi.org/10.1007/s10617-018-9201-8

[52] K. Chandramohan and M. F. O'Boyle, "Partitioning data-parallel programs for heterogeneous mpsocs: Time and energy design space exploration," in *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '14. New York, NY, USA: ACM, 2014, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/2597809.2597822

[53] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl™deep learning accelerator on arria 10," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021738

[54] G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre, "Caffepresso: An optimized library for deep learning on embedded accelerator-based platforms," in *2016 International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES)*, Oct 2016, pp. 1–10.

[55] M. A. et al., "Tensorflow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[56] C. Hsieh, A. A. Sani, and N. Dutt, "The case for exploiting underutilized resources in heterogeneous mobile architectures," in *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2019.

[57] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–10.

[58] Qualcomm. (2013) Hexagon dsp sdk. [Online]. Available: https://developer.qualcomm.com/software/hexagon-dsp-sdk

[59] N. D. Dutt, A. Jantsch, and S. Sarma, "Toward smart embedded systems: A self-aware system-on-chip (soc) perspective," *ACM Trans. Embedded Comput. Syst.*, vol. 15, no. 2, pp. 22:1–22:27, 2016. [Online]. Available: https://doi.org/10.1145/2872936

[60] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, March 2002.

[61] A. Kumar. (2008) Multiprocessing with the completely fair scheduler.

[62] Qualcomm. (2013) Snapdrgon profiler. [Online]. Available: https://developer.qualcomm.com/software/hexagon-dsp-sdk

[63] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron, "Dynamic heterogeneous scheduling decisions using historical runtime data," in *Workshop on Applications for Multi-and Many-Core Processors (A4MMC)*, 2011.

[64] H. J. Choi, D. O. Son, S. G. Kang, J. M. Kim, H.-H. Lee, and C. H. Kim, "An efficient scheduling scheme using estimated execution time for heterogeneous computing systems," *The Journal of Supercomputing*, vol. 65, no. 2, pp. 886–902, Aug 2013. [Online]. Available: https://doi.org/10.1007/s11227-013-0870-6

[65] D. Kadjo, R. Ayoub, M. Kishinevsky, and P. V. Gratz, "A control-theoretic approach for energy efficient cpu-gpu subsystem in mobile platforms," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2015, pp. 1–6.

[66] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput. : Pract. Exper.*, vol. 23, no. 2, pp. 187–198, Feb. 2011. [Online]. Available: http://dx.doi.org/10.1002/cpe.1631

[67] H. Zhou and C. Liu, "Task mapping in heterogeneous embedded systems for fast completion time," in *2014 International Conference on Embedded Software (EMSOFT)*, Oct 2014, pp. 1–10.

[68] Y. Wen, Z. Wang, and M. F. P. O'Boyle, "Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms," in *2014 21st International Conference on High Performance Computing (HiPC)*, Dec 2014, pp. 1–10.

[69] C. Bolchini, S. Cherubin, G. C. Durelli, S. Libutti, A. Miele, and M. D. Santambrogio, "A runtime controller for opencl applications on heterogeneous system architectures," *SIGBED Rev.*, vol. 15, no. 1, pp. 29–35, Mar. 2018. [Online]. Available: http://doi.acm.org/10.1145/3199610.3199614

[70] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo, "Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources," in *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '16. New York, NY, USA: ACM, 2016, pp. 320–333. [Online]. Available: http://doi.acm.org/10.1145/2973750.2973777

[71] S. Aldegheri, S. Manzato, and N. Bombieri, "Enhancing performance of computer vision applications on low-power embedded systems through heterogeneous parallel programming," in *IFIP/IEEE International Conference on Very Large Scale Integration, VLSI-SoC 2018, Verona, Italy, October 8-10, 2018*, 2018, pp. 119–124. [Online]. Available: https://doi.org/10.1109/VLSI-SoC.2018.8644937